Performance Engineering Best Practices

Building high performance cloud IT platforms using Java

Nolan V. Jones



www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65895-087

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

My beloved family:
My wife ***Gina Michelle Jones*** *and*
my son ***Dalton Vincent Jones***
and
My late dog ***Mocha***

About the Author

Nolan Vincent Jones is an accomplished IT Leader who lives in O'Fallon Missouri in the United States. He started his career in 1989 after receiving a Bachelor's of Science in Electronics Engineering Technology. His first job was working with Metrology automation calibrating and testing RF/microwave military grade electronic equipment through an IEE-488 interface using HP9000 microcomputers and HP3000 minicomputers. He went on to work at many fortune 500 companies and has worked on leading-edge information technology initiatives such as bioinformatics and space laser communication systems where performance and scale were keys to success. He had the opportunity to write high performance software in C language to test Internal Navigation Systems for fighter jets and worked on development of DNA sequencing software products used by the food industry. He has developed real time financial systems in Java, C and .NET technologies. His big picture thinking around architecture allowed him to design/develop complex real time enterprise level telecommunication software. Many years he worked in various IT domains such as enterprise architecture, operations, development and performance engineering across many industries. He has written software in a variety of computer languages (Basic, C, Pascal, Golang, Python, Java, C++, Visual Basic, Prolog, Javascript, VBScript, Rexx, HTML, C sharp and many others). As a leader, he has effectively managed over project managers and application development teams, quality teams, performance teams, and operations teams. During his long career, he has directly managed Performance Engineering Teams at 3 large corporations and built-up new performance practices from the ground up and derived the strategies. He has developed core business enterprise level distributed software and he has been trained in important management philosophies around people leadership and defining organizational business processes. His technical insights and strong leadership in Performance Engineering, which is his strongest passion, has provided a significant benefit to many initiatives.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my wife and son for continuously encouraging me and allowing me the dedicated time to complete this. I could have never completed this book without their support. It took a great deal of effort and time.

I am grateful to the fortune 500 companies that allowed me to accrue so much experience over the years. One thing I can say that over my long career in IT, being pushed into things I didn't want to do allowed me to have more knowledge than most people who stayed in the same area or same job. If you are like me and don't care about ladder climbing and just want to pursue learning and growing your IT knowledge and building great platforms, that is one thing that I would highly encourage. If you are working in development, once you are proficient there, try operations, testing, architecture etc. If you are in operations, try development, architecture, etc. If you are an architect, try pushing yourself beyond products to the enterprise scale or move into leading a product. Learn and grow and work in each domain and you will be immensely satisfied if you are this type of individual. Leadership is something that can be satisfying as well, aligning organizational structures, empowerment, people building, etc. but save this for after you have learned the other domains and have become well versed.

My gratitude also goes to the team at BPB Publications for being supportive enough to provide me quite a long time to return edits and reviews. Working full time and writing books on the side can be a difficult endeavor, especially working in IT.

Preface

This book covers many different aspects of cloud performance engineering, the importance of automation, using the right processes and architectures. It covers code, runtime environment and web optimizations and a host of other topics that will assure that you build a high-performance IT platform with a low amount of refactoring at a high velocity.

This book takes a practical approach for new learners around performance. It first introduces what performance engineering is and goes deeper as the chapters progress. It starts out very high level around performance and processes and then once that knowledge is achieved it goes very deep.

This book is divided into 14 chapters. They will cover some basic project management and development process realities, performance engineering basics, and start working into cloud challenges. Then it will start diving into optimizations to overcome capacity and runtime cost challenges associated with cloud distributed platforms written in Java. The details are listed below.

Chapter 1: Introduction to Performance Engineering - By the end of this chapter, you will be able to understand performance engineering concepts involved when creating new IT Platforms. You will fully understand the different roles and responsibilities involved in engineering fast platforms and some performance best practices required for an organization to be successful.

Chapter 2: Building a Performance Practice - This chapter will discuss the different approaches to product delivery and how performance plays a role in the product creation and why it matters. Organizational culture is discussed along with Agile and DevOps concepts around product delivery.

By the end of this chapter, the reader will understand some of the project challenges around the different methodologies and where performance plays an important role.

Chapter 3: Challenges and Realities of Cloud Performance - By the end of this chapter, the reader will be able to understand the benefits of developing new IT platforms in the cloud and when on-premises solutions should be considered. We will discuss the advantages of each and compare the different approaches. We will also discuss chaos engineering, event driven architectures and micro-services and the roles they play in performance and scale for cloud platforms.

Chapter 4: Cloud Environment and Scalability - In this chapter, we delve into important concepts in computing; Little's Law, Amdahl's Law, the Universal Scalability Law, and the art of capacity planning in the cloud. We hope this chapter provides the reader with insightful knowledge into these core theories which underpin efficient cloud computing, and guides them in effective capacity planning. Understanding the laws around capacity planning and cloud computing is extremely practical. It helps with making strategic decisions and optimization of resources for the highest throughput.

Chapter 5: Performance Automation - In this chapter, we will discuss important concepts of performance automation to assure that platforms can be delivered at high velocity with high performance quality. We will also discuss more about Service Level Agreements (SLA), Service Level Objectives (SLO) and Service Level Indicator (SLI) concepts. We hope this chapter provides the reader with insightful knowledge into how to ensure your customers receive a quality product which underpins efficient cloud computing, and guides you in effective IT platform delivery. Understanding performance automation in cloud computing is extremely practical. It helps with determining bottlenecks and driving optimization work that needs to be completed to assure the customer has a great experience and that the platform continues to have great capacity.

Chapter 6: Cloud Web Platform Optimization - In this chapter, we will introduce the reader to the important concepts of web platform optimization in the cloud. We sincerely hope that this chapter provides them insightful knowledge regarding how to assure that their customers will receive a quality and fast product. It involves employing a set of performance strategies and techniques intended to help speed up the delivery of a website's content to browsers and improve interaction times. But its importance lies in its direct impact on user experience, engagement, and conversion rates. These aspects are all critical for a successful business having an online presence.

Chapter 7: Java Virtual Machine Optimization - In this chapter, we will introduce the reader to the important concepts of JVM optimization in the cloud. We sincerely hope that this chapter provides them with insightful knowledge regarding how to optimize web services and standalone applications for speed and stability. It involves employing a set of performance strategies and techniques intended to optimize JVMs for the application profile being ran.

Chapter 8: Java Code Optimization - In this chapter, we will introduce the reader to the important concepts of Java source code optimization in the cloud and talk more about the runtime environment as well. We will go even further with more depth and extend the last chapter by discussing how code and the runtime environment work together to achieve

optimized code. Design patterns will be discussed. and we will provide the reader with good third-party sources of information around designing high performant platforms.

Chapter 9: Database Optimization - In this chapter, we are going to go deeper into the database and talk about optimization and how to make operations faster. We will discuss different types and different tips and techniques to improve performance. Optimization includes augmenting queries and tuning the database to run faster with less resources to provide the same functionality as before the optimization took place.

Chapter 10: High Performance Architectures - After discussing good design patterns for performance for the web and database, it is time to talk more about basic architecture in this chapter and how it impacts performance. We will discuss different architecture types, then talk more about what companies are doing around software creation and performance. The reader must first have a basic understanding of software development practices as a precursor to reading this chapter on architecture to benefit from the information presented.

Chapter 11: Error Handling and Logging - Now that we have come so far into discussing good design patterns for performance around architecture, it is time to talk more about lower-level implementation mechanisms supporting the product that has been developed. We will discuss different logging levels, exception handling and how to have applications log with great performance. Readers must understand Java development to understand this chapter as a precursor to reading this chapter on logging and error handling.

Chapter 12: Cloud Observability and Cloud Profiling - In this chapter, we are going to focus entirely on observability on both the front end and backend and see how we can understand what is going on from a performance perspective.

We will talk about the different cloud observability solutions that are native to each vendor and go deeper into how to do thread and stack dumps to find contention and how to do heap dumps to see what classes are on the heap consuming memory.

Chapter 13: Alerting Strategies - In this chapter, we are going to focus on implementing a good alerting strategy. Previously, in this book we talked about logging and how important it was to only log actional items. If something is being written to the log that is not actionable, then we should not be logging it. This ties into alerting. We do not want false positives with alerting due to poor logging standards and we do not want to rely on high level return codes for services or high-level application modules only without considering the anomalies that may be occurring underneath.

We talked about common sense approaches to error handling for IT support staff, determining root cause to issues when something goes awry leveraging observability and profiling. But how do we know when to alert and what to alert? In this chapter, we will talk more about alerting in general and discuss these topics along with different cloud vendor solutions around alerting.

Chapter 14: Future Trends in Performance Engineering - In this chapter, we are talking about the future and how IT platforms are evolving. In this chapter, we are going to focus entirely on the future of both front end and back-end technologies.

We will talk about Artificial Intelligence and how design patterns and frameworks are evolving. The Fetch Priority API will be discussed along with new approaches to automation in performance engineering practices.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/58fd66

The code bundle for the book is also hosted on GitHub at
https://github.com/bpbpublications/Performance-Engineering-Best-Practices.
In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Table of Contents

1. Introduction to Performance Engineering 1
Introduction 1
Structure 2
Objectives 2
Performance engineering 2
Why do we care about performance 4
Network bandwidth 5
Network speed 5
Internet service providers 6
Data and page weights 8
Performance budgets 9
Revenue benefits achieved 10
Architecture layers of complexity 11
Roles and responsibilities 15
Complexity impacts performance predictability 16
Balancing priorities between non-functional and functional requirements 16
The importance of usability 17
Bad customer experience 18
Tracking the user experience 18
Good user experience 19
Usability studies 19
Create fast products and generate more revenue 20
Fixing performance late is expensive 20
Why is performance not built into all products 21
Roles not understood in IT 22
Check in the box testing 24
Roles and responsibilities 25
Optimize first mindset 26
Engineering versus performance testing 27

Performance tester ... *28*
Performance testers need observability to profile an application ... *29*
Different performance test types ... *30*
Spike testing ... *30*
Soak testing ... *30*
Performance load testing ... *31*
Capacity testing ... *32*
Product architect/Business analyst ... *33*
Platform architect ... *33*
Performance architect ... *34*
Performance tester ... *34*
Software engineer ... *34*
Site reliability engineer ... *35*
Quality engineer ... *35*
Capacity planner/engineer ... *35*
Performance tools ... 36
Conclusion ... 38

2. Building a Performance Practice ... 39
Introduction ... 39
Structure ... 39
Objectives ... 40
Performance engineering in software development ... 40
Building performance culture within development teams ... *41*
Education ... *41*
Establishing clear performance goals ... *42*
Performance reviews ... *42*
Incorporating performance into individual roles ... *43*
Encourage ownership and accountability ... *43*
Consider performance from the very beginning ... *43*
Establish clear performance requirements ... *44*
The project iron triangle ... 45

Software development methodologies ... 48
IT waterfall software development lifecycle ... 48
IT iterative software development lifecycle ... 51
IT Agile software development lifecycle ... 52
Agile Manifesto ... 53
IT DevOps ... 56
Project delivery resources ... 57
Technological innovation ... 58
Keeping the process small initially ... 59
Staying ahead of the competition ... 59
Total quality management ... 60
IT performance cycle delivery process ... 61
IT performance cycle ... 62
Performance driven design/development ... 62
API performance level testing ... 63
Journey level testing each release ... 63
Performance code analysis ... 64
Capacity management/testing ... 64
Infrastructure analysis ... 65
Design considerations and product scoring ... 65
Site reliability engineering ... 68
Performance first product development process ... 73
Performance driven development with JMH ... 75
Load testing APIs with JMH ... 76
Performance journey level automation ... 77
Defining the workload model ... 80
Conclusion ... 81

3. Challenges and Realities of Cloud Performance ... 83
Introduction ... 83
Structure ... 83
Objectives ... 84

On-premises versus cloud IT platform development considerations 84
Cloud platform advantages *85*
Cloud performance challenges 87
Cloud managed services *90*
How different cloud components impact performance *90*
Region to region latency *92*
Resilience, failover and scale impacts *92*
Security impacts on performance *93*
Uncertainty of measurement 94
Navigating the challenges between on-premises and cloud optimization 96
Complex distributed cloud application evolution 100
Role of performance engineers and software developers *101*
Event-driven architectures 102
Micro-services 103
Adopting micro-services for cloud applications *104*
Walking the tightrope *105*
Chaos engineering 106
Chaos in action *108*
Conclusion 109

4. Cloud Environment and Scalability 111
Introduction 111
Structure 111
Objectives 112
Capacity planning concepts 112
Why optimization is important *114*
Pipeline to traffic pattern correlation *114*
Testing environment similar to production is important *115*
Understanding percentiles *116*
Design for interruption and short-term events *117*
Scaling across distributed platforms *119*
Software capacity *121*

Understanding third-party dependencies *123*
Network tuning 124
Playbook for performance best practices 125
Scaling cloud architectures 131
Micro-services *131*
Scaling databases *134*
Optimizing configurations in Kubernetes *139*
Caching *140*
Conclusion 144

5. Performance Automation 145
Introductions 145
Structure 146
Objectives 146
API and journey level performance automation 146
Service level agreements *148*
Understanding MTTR and MTTA *148*
Service level indicators *148*
Service level objectives *149*
Tooling used for performance automation *150*
Device clouds *151*
Quality gating 152
Conclusion 156

6. Cloud Web Platform Optimization 157
Introduction 157
Structure 157
Objectives 158
Identifying web performance bottlenecks 158
Web performance optimization *159*
TTL caching optimization *159*
Optimization tools *160*

Waterfall optimization ... *161*
Geo-location optimization ... *162*
Cross browser/Device testing ... *162*
Core Web Vitals and best practices for web performance ... *163*
Content delivery networks ... *165*
Always optimize images regardless of size ... *165*
Multi-layer caching ... *167*
Performing content retrievals ahead of time ... *168*
Performance design patterns ... *169*
HTTP protocols that impact performance ... *171*
Understanding rendering workflow dynamics ... 171
IFrame page load impacts ... 176
The downsides to IFrame ... *177*
Creating an IFrame ... *178*
Conclusion ... 180

7. Java Virtual Machine Optimization ... 181
Introduction ... 181
Structure ... 181
Objectives ... 182
Introduction to Java Virtual Machine optimization ... 182
Native code compilation ... 184
Removing overhead / Garbage collector optimization ... 188
Tuning web services ... 201
Conclusion ... 204

8. Java Code Optimization ... 205
Introduction ... 205
Structure ... 206
Objectives ... 206
JVM and code optimization relationships ... 206
Compile threshold ... *208*
Native compilation into code cache

Efficient data structures *212*
When classes are not thread safe *213*
Queuing *215*
Efficient exception handling *215*
Multi-threading and asynchronous programming 217
Creating fast code through best practices 220
Reusable design patterns for high performance 225
Conclusion 234

9. Database Optimization 235
Introduction 235
Structure 236
Objectives 236
RDBMS optimization basics 236
Most common performance issues *236*
Keys *237*
Basic SQL optimization *237*
Normalization versus denormalization *240*
Stored procedures 241
NoSQL databases 242
Conclusion 246

10. High Performance Architectures 247
Introduction 247
Structure 248
Objectives 248
Introduction to software architecture 248
Monolithic architecture *249*
Microservices architecture *250*
Docker *255*
Service oriented architecture *256*
Enterpriser service bus *256*
Service orchestration

Comparing SOA to microservices *257*
Event driven architecture *258*
Data centered architecture *259*
Client server architecture *259*
Layered n-tier architecture *259*
Separating batch and real-time 262
Pervasive design patterns 263
HTML 5 and local storage sync patterns *263*
Localized data security *264*
Syncing data *265*
Offloading UI tasks and past UI challenges *265*
Lightweight simplicity *265*
Cloud database selection *266*
Improving API performance 267
Conclusion 269

11. Error Handling and Logging 271
Introduction 271
Structure 272
Objectives 272
Introduction to logging 272
Error handling 273
Actionable logging *273*
Exception handling with Java *274*
Making the IT platform supportable *274*
Speeding up troubleshooting *275*
Java exception types *277*
JavaScript exception types *278*
Log standardization and documentation 279
Conclusion 280

12. Cloud Observability and Cloud Profiling 281
Introduction 281
Structure 281
Objectives 282
Introduction to observability 282
Why observability matters 282
How to find which tool to use 283
What cloud providers offer for observability 283
JMX and JTI 288
Java Mission Control 288
OpenTelemetry 289
Observability performance impacts 290
Advantages of expensive APM tools over free options 290
What do APM tools do 290
Supporting the IT platform 291
Understanding thread dumps 291
Analyzing code upfront 292
Conclusion 292

13. Alerting Strategies 293
Introduction 293
Structure 294
Objectives 294
Introduction to alerting 294
Setting up alerts in the cloud 296
Paging hierarchy/Support processes 299
Conclusion 300

14. Future Trends in Performance Engineering 301
Introduction 301
Structure 302
Objectives 302
Future predictions

Device farms *302*
Frontend performance and optimization *302*
Improvement in delivery processes *303*
NoSQL *303*
Frontend web frameworks *304*
Better native compilation solutions *304*
Self-healing *304*
Managed services *305*
Virtual IDE solutions *305*
Fetch priority API 305
Performance automation with behavior driven development 306
Artificial intelligence 311
Conclusion 312

Index 313-321

Chapter 1
Introduction to Performance Engineering

Introduction

This chapter of the book will introduce you to engineering concepts to create fast IT platforms! It will discuss why performance matters to an organization looking to build revenue off of their product deliverables. Processes, roles and responsibilities will then be discussed and considerations will be put front and center around building fast software

Unlike traditional performance testing practices, which often occurs far right in the product delivery process, well after development completes; performance engineering integrates continuous performance improvements from the inception of system architectures far left in the process all the way through to the right through design, code, testing and monitoring phases of an implementation. This integration and early considerations around performance allows for the early detection of potential bottlenecks and enables teams to make informed decisions about architecture and design to preemptively address these issues when they are lower cost to address. It also embeds checks and balances around quality early to assure that performance objectives are met when the product is delivered. Finding issues only during a performance testing process means that an architecture change can quickly become a costly endeavor greatly impacting product delivery risk.

The importance of performance engineering has never been more significant due to growing demands on IT platforms to handle increased loads and complex transactions without much degradation and uncertainty around usability. By embedding performance considerations into all phases of development,

applications that are robust, scalable, and aligned with user needs while reducing cost significantly by optimizing resources. No one wants to pay ongoing runtime costs long term due to not considering performance upfront during the build phase. This is an important chapter that will introduce you to the different aspects of performance in an IT system from tools, roles, design to mindset.

Structure

This chapter covers the following the following topics:

- Performance engineering
- Architectural layers of complexity
- Roles and responsibilities
- The importance of usability
- Optimize upfront mindset
- Engineering versus performance testing
- Performance tools

Objectives

By the end of this chapter, you will be able to understand engineering concepts involved when creating new IT Platforms related to performance. You will fully understand the different roles and responsibilities involved in creating fast platforms and some performance best practices required for an organization to be successful.

Performance engineering

Engineering fast platforms involves measuring performance early in the development process all the way through to production. It includes designing performant software architectures, infrastructure, code, databases and balancing performance and flexibility considerations in all final decisions around a platform's creation. For an organization to be successful creating responsive revenue generating products, the organization must encapsulate a great software design process. This means focusing on performance upfront and incorporating the proper coding practices to assure the platform is fast out of the gate. Many times, these coding practices are referred to as **performance driven development** (**PDD**). Beyond IT product creation processes farther left in the product creation process, focused and broad-based testing practices occur throughout platform development which includes continual ongoing platform performance analysis, iterative optimization, and full operational automation, and quality gating. Everyone should be responsible for the performance in creation. This should be more important for core features that drive the largest revenue than any other minor feature for the product.

A quote that one should be aware of:

> *If it is fast and ugly, they will use it and curse you; if it is slow, they will not use it.*
>
> *-David Cheriton, Billionaire IT Investor*

Figure 1.1 below represents the amount of data transferred to a user's browser over the past twelve years. This helps represent part of the problem that performance engineers need to solve. You can see from the chart that in 2012, the 90th percentile, meaning nighty percent of the top one hundred thousand websites existing on the web, transferred around two megabytes across the web to the user's browser on average. In 2023, just eleven years later, the 90th percentile is already nine megabytes, a 450% increase. Websites today are heavier than ever!

Now, perhaps this does not sound bad to you given the times. For many years in IT, we had the concept of Moore's law that stated that the number of transistors on computer chips and integrated circuits doubled every two years, meaning technology continued to evolve with respect to compute power. If we follow this law as a guide, we would have at least five times the compute power today compared to eleven years ago. With the evolution of cloud technology, solid state disks and hardware supporting more and more memory, this is probably right and Moore's law held true. Surely, that is good enough, pages are heavier, but compute is that many times faster as well. But let us consider what else has changed since 2012. Even if Moore's law was still relevant today, which is up for debate in some technology circles, web speed is not about just compute power, it is about network bandwidth limitations, network switching latencies, balancers with a great deal of rules to transverse and filter, reverse proxies, web and application servers, in addition to hops across all of these endpoints. In 2012, websites were hosted on a webserver and perhaps an application server interacting with a relational database. The layers consisted a **network address translation** (**NAT**) external IP to internal IP routing, load balancer to balance traffic across a couple of regions, a reverse proxy, web server, application server and a relational database. As platforms become more distributed over time with multiple cloud technologies integrating with one another, performance becomes much more challenging.

Today, we have more architectural complexity, much smaller components, many times event driven architectures with a great deal of complexity. The time it takes to retrieve, transfer data, parse and transform that data across more architecture layers, databases, disks, etc. the more potential points of slowness that will be encountered. Hence, compute may be much faster at each layer in 2023, but architectural layering, containerization and the amount of integration points has exponentially pushed upward impacting performance. Client computers have gotten faster with faster CPUs resulting from Moore's law, but browsers receiving data are still limited on threads per domain and I/O thread blocking is the largest obstacle when downloading large amounts of data into a user's browser. When it comes to designing fast platforms, we want to be in the 10th percentile shown in *Figure 1.1*. These are the fastest sites on the internet, with page weights under one megabyte.

When we have more architecture layers on the backend, and more devices or browser versions on the frontend, this is where the benefits of

important. We want to use lightweight web frameworks on the frontend, and leverage good performance practices on the backend. Beyond all else we want to design web applications using responsive web design techniques to support different viewport sizes and slim lined layouts. The same amount of data coming to a desktop application should not be coming to a mobile device. But the absolute best advice we can give is to always design with a pervasive and resiliency mindset. This means that transactions should only go across the network when required and have some level of functionality when connectivity does not exist. In other words, leverage HTML 5 technology and embrace the evolution of service workers in the browser, these two technologies allow powerful applications can be built:

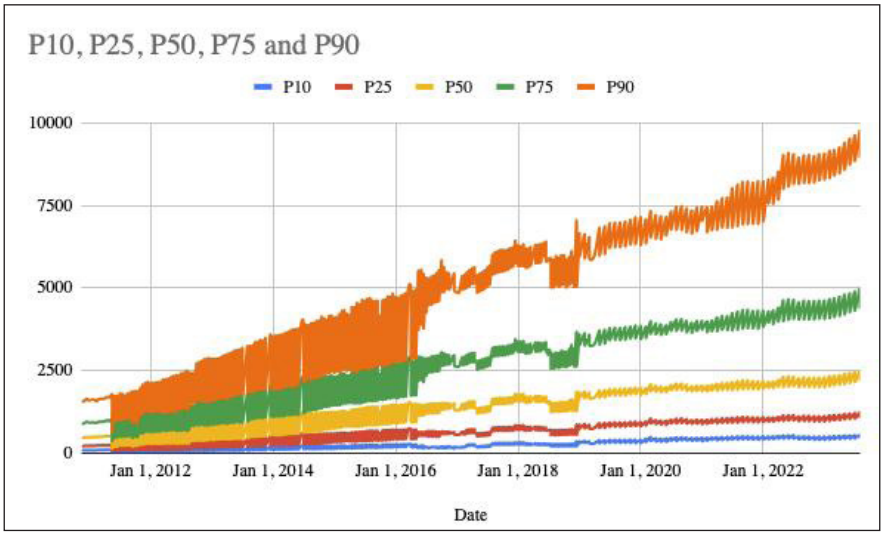


***Figure 1.1**: IT page weight trends*

Why do we care about performance

Performance has become so important in the industry today because the complexity of IT systems and the amount of data being sent over networks is increasing. Spanning from web and mobile applications to artificial intelligence and machine learning, data and software processes operating at high frequencies are traversing large datasets, which has become expensive from a performance perspective. This often includes parsing, compressing, formatting, displaying, injecting business rules, etc. while also routing across a distributed cloud and down to a user's browser or mobile application.

Search engines consider performance as one of the key metrics being looked at for **search engine optimization** (**SEO**). This is how important it is for today's IT products to be very fast. If an organization has a slow product accessed through a web interface, it will come last in search results. Unless an organization is paying a search engine provider, a promotional payment to stay at the top of search results, SEO analysis behind the scenes decides which results are featured first and which should be last below a search result.

Network bandwidth

When it comes to achieving good IT platform performance, the most important but often forgotten first consideration is fully understanding network bandwidth and how this impacts IT platform performance. Network bandwidth refers to how much data can be transferred over a network in a specific amount of time. We have seen so many performance testers and engineers run tests and never once consider reviewing bandwidth dynamics for an application at all. Architects and software designers, much less database designers, who are closest to the data, rarely think about it.

Whether an organization is operating in the cloud, on-premises, across a wireless connection, or a combination of the three, nothing is more important than understanding the dynamics around network bandwidth as the first step. If an organization indicates that a webpage takes 3 seconds the load for a website, the first thing we should ask is, at what bandwidth? If you are a Business Analyst or product architect and you define a requirement for a web service that needs to respond in 500ms, the first question we need to ask is, at what bandwidth? performance engineers should always ask about bandwidth and make sure it is defined as part of the requirement. They should know that any requirement defined for a web page load time or web service is very much dependent on it. Slow bandwidths will quickly slow down and backup an IT platform, network bandwidth variance across components and regions is critical to understanding capacity and impacts to the user experience.

Network speed

It is important to always know the network speed and network speed standard deviation with respect to packet latency. Networks transmit data in the form of packets, which are small chunks of data that, when strung together, make up a larger message of data. Why? Because network speed and packet latency impact how much data can be transferred across the network in a given amount of time, thus it impacts network bandwidth. If an available network bandwidth deviates internally in an organization based on activity during the day, performance engineers should always leverage testing profiles reflecting different bandwidths to represent the dynamic across the day ideally or as a safety precaution test at the lowest bandwidth level. This allows them to fully understand the potential impact of the customer experience and capacity.

Cloud software ultimately runs at a varying bandwidth unlike on-premises servers and local data center applications in an enterprise, the cloud itself has multiple servers, network switches, balancers, proxies, and other components operating in different regions. The ingress and egress bandwidth on the cloud itself for component-to-component interaction is dependent on the machine type, and due to this, a system composed of multiple components will have various bandwidths. Due to its usage and varying network traffic, network bandwidth dynamics need to be tested with a focus on component paths to understand the variance. Cloud services are typically accessed from different networks

with different bandwidths as well. How fast data can be transferred from one network to another or one component to another varies throughout the day, and how fast a web page can be downloaded will vary throughout the day. The more network integration points, the more variance. It is important to understand client and server component bandwidths for where code is running.

The way this is done involves using different device profiles or configurations in the testing process. These configurations dictate the upload/download bandwidth to be replicated. The throttle of the CPU and memory represents the resource limitations around a device or browser. This is very important for realistic results. performance engineers use these profiles in testing across all channels and incorporate measurements to fully understand what customers are experiencing at different bandwidth levels. businesses know their customers and they also need to know customer bandwidths when interacting with their products. If a business knows that customers are e-commerce customers that leverage a common cable connection at 5 megabytes per second upload speed and 50 megabytes per second download speed, then they are testing with an internal requirement at 200 megabytes per second at 3 seconds. Then, by default, they expect the customer to experience much more latency than 3 seconds, likely closer to 12 seconds. Unfortunately, this is rarely understood and one of the largest misses seen in by performance testing teams. The other common ignorance being the limitations around CPU on mobile devices and desktops and understanding how these dynamics impact a product's response time. Cloud resources themselves have pretty good bandwidths for high horsepower machines these days compared to on-premises solutions. The restrictions are usually the providers that businesses use to access the Internet that ends up slowing down transactions.

Network speeds can also dynamically vary based on packet congestion as far as how much traffic is sent all at once to an endpoint. For example, if requests are sent all at once, it causes TCP congestion. Networks will use TCP slow start algorithms to balance network connection speed by slowing down the connection to prevent overload of the receiver. It will eventually achieve an equilibrium between the sender and receiver and fix the network bandwidth.

Internet service providers

Users that leverage an IT platform will use service providers for internet service or dedicated lines in many instances. These providers sell connections and services at all different bandwidths. Organizations will almost always have tiered pricing at different levels of bandwidth to sell to customers while residential bandwidths can be much slower than, what is running internally at a company that has high speed needs to keep productivity high. However, the bandwidth across companies and residences varies by a great deal for both cost and network limitation reasons around hardware and software. What runs fast for one company can be slow for another.

For **business to business** (**B2B**) services, the same situation applies. Some companies leveraging a lower tiered bandwidth model through their

or a dedicated telecom phone line, will encounter a slower service response than another organization using a higher pricing tier or higher bandwidth provider.

When interacting between two cloud services, bandwidths will inherently be much different. When interfacing between a cloud to on-premises platform, bandwidths will change. It is always important to consider bandwidth across all interaction points and understand how fast data is transferred between networks for both requests and responses (uploads and downloads) and consider this in testing results.

When two systems communicate, these packets send a request to inform the other that more information is coming, after which responses are sent back. The important thing to remember is that if it does not get a response, this will sometimes cause a packet to re-transmit. Resending the packet, a second time will cause latency across the network. Ideally, retransmits should stay under 2%, or network teams or cloud support teams should be engaged to see what is causing the retransmits, which may be a faulty switch or other network component.

Over the years, network and ISP bandwidths have continued to increase to help with customer demand for performance. Mobile has reached 5G speeds and is faster than it has ever been in the past. Traditional telecom network topologies can be complex, but the hierarchy is generally multiple central hubs exist in countries that route the traffic to localized locations closer to homes and businesses. These remote locations are referred to merely as nodes. Fiber optics are used to deliver high bandwidth to this central hub or very large bandwidth cables, some laid on the ocean floor. Fiber optic cables are used to communicate to a node. The central hub fiber optic connection to the node is officially called **fiber** to the node by telecom providers. The node then sends traffic to residence or business customers. One of the largest initiatives we have ever seen in that industry was moving away from the large old copper cables and moving to fiber optics to allow for more bandwidth to be transmitted closer to business. It literally is what kept many telecom providers from going out of business. Wireless technology and cable providers were taking away all business from the traditional telecom solutions and this was their lifeline. Fiber optics are one of the most efficient ways to transmit and receive high speed data with less signal loss than traditional copper lines that were leveraged for decades for landline phone communication.

In telecom, most of the path traveled for data is using fiber optics allowing for greater speed and performance as far as sending data around the world. Where the solution has historically fallen short, compared to many local traditional cable providers who run high bandwidth cables for television services, is that old and new technologies are combined in the final solution. The node that received information from the central hub may have been high speed, but many times when the node sends traffic to the residence or business, it is still leveraging old copper phone lines. So, fiber is not always end to end. When network traffic goes from fiber optics to copper, this results in an excessive amount of signal loss and some noise ends up on the communications electronic signal. The signal itself is attenuated rapidly based on the distance the residence or business is from the node. The

closer the physical location the smaller the loss and the better the signal. This is typical of **Digital Subscriber Line/Loop** (**DSL**) solutions that, for years, many people had in their homes. Due to this loss in amplitude from going from fiber to copper, the distance between the node and the residence or business is limited to a short distance. Newer solutions have been more advanced to supply fiber to a residence or business. This, and wireless technology that is limited with line-of-sight towers around the globe, has been the future to bring high speed internet to all locations in every country.

Data and page weights

However, even with more bandwidth available to businesses and consumers, businesses themselves have pushed software development and product creation to move at a faster pace. This has resulted in the simplification of development practices using higher levels of abstractions around source code using much bulkier frameworks than ever in the past. The larger the framework and more abstractions incorporated, the slower the code. Codes with other code wrapped to simplify it, ends up being slower to run than code written at a lower-level method that is lighter weight. Some new computer languages have attempted to augment this and create products that run very fast but are written in fewer lines of code, like Rust, Golang, and others, where you have simplified development practices for rapid product creation and still fast speed. Many organizations leverage C# or Java languages to build products because of the third-party support, even though these languages are not as efficient.

Some organizations have moved to create products in JavaScript using tools like Node.js, which is a very fast tool to retrieve data from a document NoSQL database, which can send data back to a browser in JSON format to be parsed and displayed. This solution is great for simple tasks with just retrieving data from a database and can create some very fast websites, but it bottlenecks on anything computationally intensive or where much parsing is taking place due to slower code interpretation at runtime. Supporting it in production with the debugging tools can also be more challenging than using languages like Java. We supported Node.js for a large restaurant chain in the *United States* and when it worked, it worked great. When something went wrong, it took months to fix and identify the root cause. Not with the code, but with the runtime environment and thread interpretations and debugging efforts. We are sure it will evolve even more over time. It is a very fast way to create simple services or web applications. Golang, Rust, C, Java, .NET are more robust solutions and finding root cause with these will be much faster.

It is also very easy to find large vendor resource streams onshore and offshore that can easily create software rapidly in these languages over Golang, Rust, or C language. Popular Java and .NET technologies not only leverage an intermediary virtual machine over the native operating system, they also leverage high levels of nested code over C language code that runs the virtual machine runtime environment. So new frameworks end up being code wrapping code wrapping code wrapping code, and so on. The more layers of abstraction the easier and faster it is to write programs for product development, but the slower it is to run the code using these languages. The same goes for large web-focused JavaScript

frameworks like AngularJS and other comprehensive web frameworks, which are many times larger than if programming to a much lower level where most websites only use around 20% of the framework's capabilities.

`Webpack` is a module bundler that helps optimize deployments and helps split out JavaScript code into modules that make it load quicker, but the Angular framework can be very slow. If the many optimization features are not leveraged in AngularJS, it can be unusable. A whole new market has opened with tools on the frontend with web frameworks like React and Vue which are lighter weight, have come into play to combat the performance issue with Angular and allow web elements to be manipulated in memory called a **virtual Document Object Model** (**DOM**) versus the traditional DOM. Whether it is code downloaded into a browser or loaded by a runtime class loader process, the more code layers in Java, or in any language, the more latency to be expected. Also, processing threads in web browsers, overly complex filters or rules in balancers and reverse proxies and other runtime system complexities limit the performance and continue to grow in the cloud. For example, refer to *Figure 1.1* above which reflects the size of web pages in IT platforms being sent to a user's browser over time in kilobytes. Page weights reflecting the amount of data have gone steadily upward in the past 10 years.

What happens when page weights or data being sent across a network goes upward? Performance inherently quickly goes the other direction, downward, it has an inverse relationship. Another key performance metric measured during testing or analysis is called **processing rate** or **throughput**. As page weights increase, data transfer sizes increase, response times thus increase and processing rates and throughput decreases. Thus, processing rates and response times are very much an inverse relationship. You can tell when a platform is reaching its capacity when you start to see response times increase at a given processing rate. The processing rate or throughput which are uniformly the same term, determines how many requests can be handled in a period. When an IT platform is tested, response time will remain relatively flat for a while and when processing rates start to decrease as more requests go into the platform under load, response times go up. At this point, generally, the platform will tend to resist any more additional requests; transactions take longer and longer until, eventually, the platform stops being able to respond with the needed results or exceeds the **service level objectives** (**SLO**) where the platform is considered still usable. Hence, the amount of data, processing rates and response times all have a relationship that is important to understand. These are just some of the metrics to really get a good handle on to understand performance.

Performance budgets

As we dive deeper, one important concept is understanding how a performance budget works. This is not referring to expenses around performance, it is referring to tradeoffs between latencies around components and services to achieve an end state goal. If we need a platform requirement to be able to process a transaction in 4 seconds and the transaction consists of 4 requests across the network to a single service or multiple services, then we can inherently make tradeoffs around what to

can have one request be 500ms and another be 3 seconds, and two be 250ms each. We could also have one request be 1.5 seconds, another be 2 seconds and two be 250ms each. These tradeoff dynamics reflect flexibility around optimization, which allows us to reach our end-state goal, and thus, capacity and user experience requirements end up being fulfilled. This is a very important concept not always understood. You do not need to set requirements around individual requests with respect to latency as much as transactions composed of multiple requests to accomplish a goal.

The goal around performance is always about two things: a great user experience and capacity. One could also argue that runtime cost is very much a strong third when it comes to cloud costs. These dynamics can also be translated to web page load times. If a requirement is to load a web page in 3 seconds over a 200 megabyte per second connection, then the individual requests across the network to load that page are all up for grabs around optimization to achieve the performance budget of 3 seconds. It is always a good idea to set guidelines around best practices far left in the product creation process around request latencies, but as far as an actual requirement, business transactions should be the focus and flexibility is around optimizing what is necessary to meet the requirement. The reason that this is so important is because of how performance ultimately impacts revenue with respect to runtime costs, usability and brand perception.

Revenue benefits achieved

Some examples over the past 10 years where organizations saw revenue or cost saving benefits from performance optimization:

- **Bing**: A page that was 2 seconds slower resulted in a 4.3% revenue drop
- **Google**: A 400ms delay resulted in 0.6% drop in searches
- **Yahoo**: A 400ms slow down resulted in 5-9% reduction in full page traffic
- **Shopzilla**: Speeding up their site by 5 seconds increased the conversion rate 7-12%
- **Mozilla**: Shaving 2.2 seconds off their landing pages increased conversions by 15.4%
- **Netflix**: Adopting a single optimization, gzip compression, resulted in a 13-25% speedup and cut their network traffic by 50%

The complexity of large, distributed cloud-based systems and the amount of software and infrastructure components interactions all play a major role in poor performance and standard deviation around response times being seen on IT platforms. These are just the tip of the Iceberg when it comes to why good engineering is so important. Engineering platforms for good performance is not just about running at low cost or capacity, it is about having a platform that is usable where perceptions of the brand for the organization becomes a positive experience.

Consumers using websites today have high expectations when it comes to usability and

from using an application. **Bounce** is an industry term to mean that they stop using the application at a point in time when going through a user journey in the application. A growing trend over the past decade in IT has been that platform loyalty is very limited. Consumers today will ultimately not put up with a slow website or mobile application over time because they do not need to. This is especially true for younger generations where speed is considered the norm when it comes to website interaction and the expectation of instantaneous interactions. If you do not have a fast platform, customers will ultimately drop your brand in favor of the competition who has a faster implementation or has that perception of quality before you realize it. Even if you released a new IT platform rich in features with the best of intentions, if that functionality is slow, you may have lost a customer vs. gaining or keeping one which was your original intent. Some of this is all due to brand perception and not always about usability, it can be about perceived quality and mindsets.

Architecture layers of complexity

Modern website servers both in the cloud and on-premise infrastructures are under high load today and consume unprecedented amounts of raw computational power due to growth in the mobile space and the connectivity now available to people around the world. IT teams involved with software development, capacity planning or IT server operations are familiar with this issue. But with great power comes an even greater need to assure that an IT platform is responsive, can scale well, and is highly resilient in nature. Obtaining good performance results for an IT platform these days requires a comprehensive understanding of all layers of the underlying platform, deep insight into the processing at hand, and the ingenuity and creativity required to obtain effective performance across the many layers on each piece of hardware. It is important to understand what needs to be tuned, what needs to be cached, and what needs to be distributed or parallelized from a computational perspective to achieve the balance needed for maximum throughput.

The reward for mastering these sophisticated and challenging goals is the ability to make web interactions that can process large amounts of transactions that are orders of magnitude faster and more efficient than if proper performance processes were not put in place. This allows IT platforms to obtain results that have been not possible in the past with single non-distributed processing. Poor responsiveness and lack of scalability can make or break a web software-based product. The advent of rich content has increased page download time latencies and decreased bandwidth availability. This along with greater processing power needed by the web has pushed sites to look for tuning options to build additional scale. In most cases large software architectures today have layers. These layers distribute computational processing across multiple machines.

As loads increase for a software product, latencies across the many layers can begin to back up to the point of kicking out users to the IT platform. For example, architectural layer one will spin off a threaded connection, which calls out to layer two which calls out to layer three and so on in a distributed computing environment. If layer three starts to backup,

then layer two starts to back up and eventually layer one. If this happens for a long enough period, the entire platform will no longer be able to respond. Tuning these layers to make sure layer three can respond quickly to layer two and so on keeps the platform responding to requests from the client web or mobile application in a timely manner. This is important to achieve maximum throughput and responsiveness for a website.

The heaviest load in any software architecture is where the most time-consuming processing is taking place. To get layer two and layer three to be able to respond as to the faster level one layer, caching must be put in place to reduce the computational processing on the servers. The goal is to not repeat processing that has taken place already and have it available in memory allowing the request to immediately come back to the calling layer versus adding additional computational load on the servers thus encountering additional latency for the request. Sometimes threads can get hung for one reason or another, a database table lock, a thread contention scenario or just very slow requests due to improper database tuning or just plain lack of computational horsepower. To keep a platform tuned optimally, timeouts should be set across the entire stack for the web platform at each layer to assure the backup of threads under load does not happen before it needs to. A well-designed platform properly tuned with the right throttle (request pool restrictions) cannot be overloaded, it may not allow more requests, but that backend infrastructure will not crash or fall over. When a platform is not designed this way, components start crashing in an unpredictable fashion at different states of operation making recovery very problematic.

The largest impacts to failing to meet performance objectives for an IT product are:

- Schedule delays
- Project cost overruns
- Project lost productivity
- Damaged customer relationships
- Missed market windows
- Lost revenues
- Technical debt

Delivering software that has both great performance and availability for core features is important. Moreover, the cost to deliver performance is higher the farther back into the process it goes. The largest and most severe performance problems are not always a result of bad coding. They are most often the result of poor architecture or software design decisions that could have been averted early in the process. Sometimes the proper separation of concerns is not followed from an architecture perspective, or the requests are too granular across the network introducing more network latency and variability in response times. Establishing a skeleton architecture representing a minimalistic design as a starting point and the doing a proof of concept around performance is important. Organizations that do this early in the project lifecycle save a great deal of cost and headaches down the road. It is best to not wait until the end to determine that you have a poor architecture or software design.

Performance cannot be retrofitted and must be designed into the software from the beginning to get good results. The retrofitting approach where performance is after afterthought and code and infrastructure is changed at the end tends to yield small wins at the expense of poorer performance design patterns causing more deviation in response times and less resilience. Architecture and software design are the most important components for implementing a well performing web platform to keep costs low.

Let us look at steps required in these product development phases to assure a well performing system:

Architect and design a flexible software architecture upfront to meet project performance objectives. Software architectures themselves should not be entirely iterative. Organizations should code and do detailed designs iteratively in an Agile process, but high level software designs and architectures should be established upfront with more focus time allocated along with design patterns and web standards to be used in the implementation. This includes tooling considerations. For example, if you are going to design software that needs to meet accessibility requirements for standard accessibility readers, use UI libraries and tooling that gives you this accessibility during the development phase as a requirement. The selection of tools and processes along with coding standards, web standards, accessibility tooling including testing tools should be defined upfront and time should be given to do proof of concepts here to make sure the right tools and processes are chosen. This is one area where waterfall and even iterative processes have worked much better than Agile. In Agile, remember to spend time early in the process coming up with the architecture, design (including database design) and do the necessary proof of concepts to understand the performance and resilience dynamics. Go slow here and then faster once you are out of the architecture and design phases. Balance flexibility with performance. The largest impacts to Agile based platforms are that architecture, designs and even requirements organically grow. This is their largest downfall when it comes to quality and maintenance costs after delivery. Instead design for architecture for flexibility upfront and identify the design patterns and standards that will be followed ongoing for product delivery. Take the time to create the needed interfaces or contracts and defined payload standards making sure that implementation code is pluggable, supportable and assures minimal architecture changes in the future. A good architecture is all about flexibility and considering the future what if scenarios upfront and not about starting small and creating something organically grown into a snowball that is unsupportable.

Identify stories or use cases that represent core critical functionality early in the iterative development cycle and associate performance objectives with each story. These are the most used journeys that will be leveraged through the user interface or backend services. These are the most important journeys in the product as far as how fast the services work, how easy they are to use and how to design them for users to spend minimal time to achieve the task they are trying to achieve. Review each user journey defined and plan to optimize software and hardware for the best response times to achieve a SLO.

Model architecture on a small scale and mockup and create a realistic proof of concept with skeleton code to understand behaviors and

stubs or interfaces with the minimal functionality to represent a working service or user interface. These allow you to see how many requests are going to be sent across the network and how large the requests will be, how the SQL in the database performs and to gauge performance bottlenecks like mapping relational database objects to Java objects and mapping Java objects to JSON. To be clear, one of the largest bottlenecks in distributed platforms is network latencies, resource constraints, the amount of data transformation that takes place and the latencies associated with data transformation. The less data transformation the faster the requests and more the platform will scale. This is one of the primary reasons NoSQL initially became so popular, it eliminates not only slow relationship normalization associated with multiple table lookups, but it also more importantly helps with eliminating data transformation. We can use NoSQL document databases like MongoDB or others to store results, then push those results right to the browser for interactions without any large data transformation work happening in the middle tiers of the product architecture. Data transformation is very CPU intensive.

1. Measure latencies coming into your infrastructure to best understand the DNS, SSL latencies and expected responsiveness of current websites. Also, include latencies around balancers with expected rules/filters and **application programming interface** (**API**) gateways with edge caching from content delivery networks. If you do not know, assume 200ms for DNS lookups and 400ms for SSL impacts and design around those worst-case numbers.

2. Adjust software and hardware architecture changes to optimize the solution based on the proof of concept results before moving forward with product development. Designing for performance is iterative in nature but the important thing to consider is that you need an initial baseline to understand if you will meet your objectives for performance. That baseline should be optimized as part of the proof of concept and then further optimized iteratively as code is delivered.

3. Design mockups early, measure, adapt infrastructure and software designs, implement code, do code reviews looking for performance anti-patterns which impact performance, measure performance, analyze web vitals and frontend performance best practices, repeat. The goal is to not only set performance goals for each use case but to iteratively put measurement of performance into your process and continue to adapt the software to meet those goals. Measure performance early and often is recommended once software is in a lower environment, this does not necessarily mean a load test.

Introducing proactive performance objectives into the **software development lifecycle** (**SDLC**) process is simple and reduces project risk considerably. First, you define the requirements for a particular iteration of development with defined features, including performance requirements, as part of this process. This should include not only throughput metrics but responsiveness metrics from various locations where the software will be used at different bandwidths and with different devices and browsers. Then, you architect, design, and code the software to these specifications. To validate the requirements, it is

important to test performance and test functionality locally to make sure the software is meeting the defined requirements. Then integration performance tests need to be performed, globally or domestically through the regions of usage to verify requirements that are still being met with an infrastructure configuration and data footprint that matches production. Functional and performance load tests from various geo-locations, devices, browsers, and bandwidths are required. Perform a resiliency test of the infrastructure and introduce various failure use cases, and understand how performance and functionality are impacted for real world events such as maintenance windows or a momentary outage. Many times, this is referred to as **chaos engineering**. Then once all issues are addressed, release the software. The goal of chaos engineering is to still meet the SLO even when under duress and have enough threads and capacity to handle the disruption to the IT platform.

Generally, it would be best to iteratively measure your website performance locally and through continuous integration channels as the software continues to evolve. You want to be able to track performance changes over time. Beyond incorporating performance in every stage of the SDLC, it is important to utilize performance driven design best practices for software design that are discussed later in the book. To achieve the best results with respect to understanding what your platform can handle, create a performance environment that matches your production environment with respect to hardware, software, and data. Take a snapshot of the data in production and move it back to your staging or development environments for real world performance measurement and a recurring basis to measure how the platform is changing with respect to being able to handle load. Then pursue the ability to test your site from various geo-locations and devices during each testing cycle. Anytime you are a part of a project, whether you are a project manager, business analyst, developer, architecture, or leadership, you play an important role in performance.

Roles and responsibilities

There is not currently a single role in any large enterprise that can achieve what is needed to assure good platform performance in today's distributed cloud platform products. Instead, it involves all groups in the IT organization involved in the platform's creation. Good practices around performance lead to fast IT platforms. Many times, these platforms are very complex, distributed IT platforms in the cloud that leverage web pages, web services, large data batch processes, databases, balancers, reverse proxies, API gateways, network switches, etc., and may even interact with third-party integration, touch points and other clouds adding to the complexity. To achieve great platform performance, all these software and hardware components need to be synergistically calibrated for high performance and optimized to achieve both scale and usability. But to that end, the platform must be architected and designed to support great performance upfront. Tuning something coded or designed poorly is not productive and the outcome is very limited with respect to results and in most situations trying to tune a poorly coded or designed platform causes even poorer optimization practices to be followed that ultimately impact

Making changes late in the process is far too costly than considering performance upfront. Decisions need to be made around third-party dependencies and how backups downstream which a platform may have little control over impact scale and throughput upstream and designing software to account for a predictable level of backup. Also understanding how software interacts with databases, how slow database queries impact database pooling, connection acquisition, and component throughput can become a complex science in itself as components grow and databases become more distributed and shared.

Complexity impacts performance predictability

Engineering platforms for great performance have never been as complex as it is right now. The cloud expectations were that the platforms would have unlimited scale and be self-healing. However, as with all complex platforms, performance and resilience become difficult to predict, and more uncertainty exists. The more components and variables the platform has, the more complex the design and the more difficult it is to achieve consistent performance and resilience and recover from failures in an environment. Cloud providers have been quick to provide availability numbers due to redundancies but give themselves large leeway when it comes to performance numbers because of the cross region and cross component infrastructure dynamics with many layers and rules in the process from the **web application firewall** (**WAF**) to the API gateways, to the reverse proxies to the downstream app servers and databases. A lot of variables to consider and the more complex a platform because, the more difficult that task. Entering the realm of the cloud when it comes to predicting performance can be daunting, but incorporating the right practices and processes, along with some good mathematics, can make it possible.

Balancing priorities between non-functional and functional requirements

When building IT products from scratch, nonfunctional requirements matter more than functional ones in many cases with respect to revenue. I know that this statement in itself can be controversial for businesses focusing on pushing out new features over their competitors. But, at the end of the day, customers want a fast, clean and simple, easy to use, well designed platform that is defect free with core functionality and they want it to be resilient and accessible. At a minimum, this will build a good brand and assure happy customers ongoing. This should always be the goal. If you have a website that downloads into a user's browser in 6 seconds vs. 2 seconds, though not always realized initially, if you were to take a survey of customers using the platform, you would likely find that it really matters to their perception of an organization. You need to be faster than your competitors at a minimum.

When it comes to brand perception, organizations monitoring customers coming onto the platform tend to see higher bounce rates where the latency is high for user journeys in a platform or where journeys and functionality is complex making processes slow overall.

Simplicity and performance are preferred over more features that are slow and unreliable. Customer frustration is much lower. An organization that focuses on excellence in product quality will do well over the long term. Users for an IT platform will not want something that is slow to retrieve content or data from, complex, or unreliable. As a customer, would you want to use an IT platform that is slow with poor usability cluttered with features that are hard to understand requiring a learning curve or an easy-to-use core functionally streamlined product that screams with speed? You will likely lose many more customers if the platform is slow vs. if it is missing nice to have features because the nice to have features impact a small number of customers in most cases and all customers are impacted by slowness and difficult to use platforms.

The importance of usability

If you go back in time with your thoughts, think about how *Apple* and *Google* became so successful. Was it because they created IT platforms with more features than the competition? Absolutely not, just the opposite, it was because they focused on core functionality, simplicity, quality and speed! They did what the others were not doing, going simple and fast while building up a brand perception around quality and reliability. The more complex a software design, the more features, the more hardware, the more hops across a network, the more lines of code, the more errors, verbose logging, the more difficult good performance becomes and thus the practices around performance had to change to obtain and predict it. It is a brave new world in the realm of understanding all of the dynamics around performance and resilience. When *Apple* came out with the *iPod* there were hundreds of MP3 players on the market with some great designs that were not as easy to use as they could have been. But *Apple* came out with one that was fast to interact with, it had a simplistic design with great quality over the competition and beat out the whole market which shows that quality, simplicity and the perception of speed matters.

The *Google search engine* when it first came out it was < 512k of data transferred across the wire from a web server to the browser. After caching < 50k was being transferred for subsequent visits. Low page weight like this translates to extremely fast performance. They used long cache times to assure that first page load time was always kept to a minimum. At that time almost all bandwidths being used were much slower than today's standards. Consumers utilizing the search engine still managed to see sub second load times. Even today at a size of 1.1 megabytes of data transfer on the initial load to the user's browser, after caching *Google* sends a mere 72k in size for subsequent calls across the wire once caching kicks in with much less transferred. When you can keep data transfer sizes this small it means that the website will load very fast and with today's bandwidth, *Google* still manages to impress with its lightning-fast load speeds. The goal for any software design should be to have the first page load fast regardless of caching and be small in size with requests not blocking the browser threads. This is not the time to execute long database queries before rendering and loading a web page. Keep it light!

Bad customer experience

Let us look at another relatable example from the perspective of a consumer. Let us say you want to order a pizza. You search for a local pizza parlor and come across the website for *Joe's* pizza. You navigate to the website and it takes 15 seconds to load because it has a lot of large highly graphical images loaded showing the steam coming off the pizza to market to the consumer. As someone who worked in the food industry in IT, we can tell you that steam on images is a great marketing ploy to make you crave the food but it takes a lot of pixel density meaning large high-resolution images in many cases. You notice that when the website loads, it has a menu that has every combination of pizza you could think of, but it takes you about 15 minutes to understand how to order the pizza. Due to the large images, everything seems slow and interactions seem sluggish. When you select an item, it takes you to another page and takes 4 seconds to add each topping before you can check out. Due to the slowness, you feel the site is hard to use and you get the impression that *Joe's* pizza may not be a good place to put your payment information into. After all, what happens if the payment timeout is your first thought and you must keep hitting retry. Plus, more importantly at this point you feel frustrated, you were hungry and now you have had to use something that was not a good experience, and it has taken much of your time. The impression of the company is low, and you do not have a positive experience. Do you think that, if you had a choice, you would use *Joe's* pizza again?

Tracking the user experience

How many organizations do you think actually track time to do tasks for features beyond granular transaction or request types of latencies? It would be a valuable thing to know, would not it? This, along with understanding the time thresholds associated to customer's bouncing off the platform. Organizations generally do this with analytics tooling such as *Google Analytics*, *Adobe Analytics* or other similar products. These tools are used to see what customers are doing on a platform and how long they are in sessions interacting with the platform. That total session time metric in these tools, by itself, may be enough to help one understand if the platform is easy to use. That being said, a lot of good analytics solutions are used to track this type of thing including **application performance monitoring** (**APM**) tools that have **real user monitoring** (**RUM**).

APM and analytics tools provide the ability to add a JavaScript tag or snippet of code into a website's HTML source code so that user journeys through a website can be tracked. The code generally sends information to the APM or analytics tool server for tracking and sending timing information using the browser timing APIs or measuring interaction times. So, for example, if we login, then navigate to an e-commerce website, add a product to the shopping cart, checkout and pay for the product, they track our journey throughout the whole interaction and see everything that we would click on. They know how long we paused, how long requests took and if we dropped out of the application altogether. Vendors such as Datadog and Dynatrace are near the top in the industry at the time of this book's writing for APM tools, but there are many

give you the ability to profile applications, see users' sessions and track or record them, determine root causes around performance issues in the code and runtime environment beyond observability. Without observability, you are just guessing what the problems are when it comes to performance. Testing tools that generate load against a platform and record response times are not enough. You need to be able to see what is going on at every layer from the testing infrastructure to the backend services being tested. This also means analyzing and understanding the latencies associated to the testing product and its infrastructure.

Good user experience

The next time you order a pizza, you decide on a different local pizza vendor to see if they are any better than *Joe's* pizza. You do a *Google search* and find a website named *Speedy Pizza*. Their website takes approximately two seconds to load and instead of a menu with a lot of hard to understand options with multiple pages to navigate to, it has a simple wizard that streamlines your order process from start to finish from a single page. Toppings can be added to the order in less than 500ms. It has images and, though you cannot see the steam, you still see a pizza image that looks appetizing. You feel comfortable that payments will go through without any issues or timeouts because the site feels reliable, fast, and usable. Because the site is fast and feels solid, you get the impression that *Speedy's Pizza* is a quality organization that is professional, and you do not feel that you'll have any problems paying for the order or executing transactions on their website. You have your pizza ordered in 2 minutes.

Which site do you think more highly of at the end of the day and which site would you visit more frequently? It is always a balance of course between functional vs. non-functional requirements but remember that the non-functional ones are always required with core functionality over nice to have features; differentiating between core and nice to have features is important. Very fast, and easy to use products leave a lasting impression on customers and business partners. We cannot count how many times another company has reached out to me to ask about a platform's performance before doing business or scanned a site and measured it themselves.

Usability studies

How can an organization determine what really matters to their customers when it comes to usability and performance before a product is created? The answer is usability studies which are a great approach to understanding customers. Many large organizations will have usability teams that will sample the customer base, record videos of application usage, take surveys, and have them run proof of concepts with various user interfaces or interactions and performance. They run a study around platform perceptions and usability which includes surveys and more importantly recording user behavior while using the application undetected or distracting from the participant. It is easy to do a proof of

great performance vs. slower more cluttered UI and performance to see the perception of the customer before creating a product.

These proof of concept product creations are rapidly put together with high level computer languages or products; they make sure that you are really creating the right product and focusing on the right things. It is not a production ready implementation; it is a website with the interactions and small local data sets in most instances. There is nothing more important than understanding the customer that you are selling to. Try having sites at different speeds with different features and understand the perceptions from a user perspective of an application and it can be surprising how a product is perceived. Provide them a fast streamlined site with just core features and a slow multi-feature site and understand the perception. Ultimately platforms that are not reliable or fast and easy to use translates to revenue loss.

Create fast products and generate more revenue

What's strange is that still in today's world performance is an afterthought or put secondary to functional requirements when platforms are initially developed. But we have seen that change considerably over the past 10 years. Companies are learning more about how their customers behave with respect to performance and are beginning to understand frustration levels. Thus, dedicated performance teams are formed for this purpose, or the performance ownership is put into site reliability teams. But as a guideline, when should we start thinking about performance? The answer is in the requirements phase for product development. Trying to react to poor performance and rearchitecting or patching a solution at the end is much more costly to do than at the beginning and with much better results. It is really hard to beat a competitor's website performance if you have not thought about it from the very beginning. Writing a platform very fast without considering performance, then reacting with sub-par measures to shore up poor performance after the design and coding is complete, limits the achievable performance and costs much more than if it was considered from the beginning. Though the business stakeholders may define the performance requirement, the real requirement is your competitor's product or the mainstream internet latencies seen at **https://httparchive.org/**. You need to be faster than these. Always question the business stakeholders if these datapoints are not being considered for the requirement.

Fixing performance late is expensive

Having a poor user experience due to not considering performance dynamics is a very bad state to be in for a new product and the cost to fix it later vs. earlier in the process is much more significant. Think about it from a process perspective. If you find something slow late in the process, it becomes a support item. Let us say you find that the design was slow because it was doing fine grained decryption of fields coming across the wire vs. compressing and decompressing the entire payload. The time to rearchitect the application,

update old records and change the data transfer objects to support the new approach is a very long endeavor.

If you had caught this flaw earlier or had a requirement around performance with respect to the feature in the very beginning around performance and it was tested early during the design/development phases, it could be fixed there and then during product creation. Maybe during the architecture phase, it was decided to use a data store that required multi-tenant encryption but a database did not support it, so the encryption/decryption has to be done using separate libraries, thus was not seamless requiring much more latency. These situations are common ones and ones that can be avoided by thinking about performance upfront far left in the process when these things are simple to fix. This hard lesson is why the demand for strong performance IT consultants has increased over the past decade. It is important to think about performance upfront and all the way through the development lifecycle to achieve great usability, great brand perception and low runtime cost. Requirement phases define the performance, architects architect to the performance requirements, optimize their code to the performance requirements and performance/quality testers assure that the requirement has been met before releasing a product into production.

Why is performance not built into all products

Well defined performance processes are not incorporated in IT platforms due to lack of knowledge and platform complexities. It is many times just a simple matter of priorities around what a business considers important. Moreover, leaders rarely understand how slow platforms impact revenue for an organization. Unless contractual **service level agreements** (**SLA**) are signed, many times performance becomes secondary to other features. Revenue loss is not understood until further down the road when customers start moving to a competitor. For web platforms, simple low hanging fruit for performance like optimizing images regardless of size can triple the speed of many sites or splitting out JavaScript packages that are bulky on first load. Focusing on above the fold content which is the content before doing any scrolling can significantly improve the **Time to Interactive** (**TTI**). But even these things are many times not considered unless there's a noticeable lag due to excessive thread blocking in a web browser slowing rendering times. To achieve the desired performance outcomes for products, optimization efforts should be embedded in the process, it should not be reactive for just a long pole in the tent or because something was perceived as slow late in the process. It should have never gotten that far into the process in that state to begin with because it is cheaper to fix earlier.

Beyond the cost discussion, optimization is not something that is an afterthought when something as perceived as slow, we want something to be faster regardless, so we optimize everything as part of the process initially far left in the product development. For example, if a product has a 100k image that is on a website, we can optimize it and make it 50k in size. What is the impact? It means that it will likely download twice as fast from the web server to the user's browser improving page load times. We do not need to only consider the one large image that we are seeing, as a practice we

do not want to be reactive when it comes to optimization which is a trap that can be easy to fall into, instead we optimize all images, all JavaScript, all **cascading style sheets** (**CSS**), all Java code, etc. As we discussed, performance and usability impact an organization's brand which translates to business. Fast web products inherently are perceived as quality while slow sites give the impression of low-quality and something that will be unreliable. When a customer does business with other partners, they need to perceive quality which means they expect a platform to be high performance, highly usable with the core features they need.

Beyond B2B scenarios and business partner perceptions, **business to consumer** (**B2C**) users leveraging a consumer-focused website will only do so if they perceive they are dealing with quality and someone reputable. A website that takes 10 seconds to load provides the perception that it is a low-quality organization to do business with. Cost is also a factor, but it is very much a balance. Fragmented understanding around responsibility and ownership for performance has become further problematic with performance really being everyone's responsibility in IT. Attitudes such as it is not my problem and focusing only on features are more important than non-functional requirements can easily become the mantra of an organization not fully understanding how performance and usability impacts revenue. It can quickly become a check in the box for a halfway load test that is performed and nothing more until it is too late and customers abandon the product. Performance and usability should be considered the most important features for a platform. This, many times, is overlooked.

Roles not understood in IT

Let us say that you are brand new employee hired onto a performance team within an organization. So as a first step you reach out to your comrades in IT and ask them your role in the organization. What is the process for making IT platforms faster in the department? I suspect that if you ask three different people working in the field of **information technology** (**IT**) in the organization what achieving good performance entails, you would get three completely different responses. To improve performance, is not that just testing the cloud software looking for slowness one individual may say in response to your query. Another may say that you need to throw more hardware at the platform. Yet another may say, to achieve great performance you must execute capacity planning. Another, more technical person, may indicate that it involves Amdahl's law or Gunther's universal capacity modeling dynamics with queuing patterns; that perhaps these dynamics somehow translate to a full understanding of platform performance dependencies. These capacity laws, along with Little's law, are important relationships to understand around any platform's capacity which we'll dig into later. Some organizations may have a mere testing mindset, perhaps may believe that improving performance means testing associated with a release cycle, looking for the big red flags that may correlate to limits in scale. But why do people in IT think differently when it comes to performance and the role of performance improvement? Why do some organizations see performance as an afterthought, assuming

As IT professionals, most of us have a very good and unified understanding of the differences between engineering and testing skill sets and how they differ. But many of us have a difference of opinion around the definition of performance testing versus engineering and what it means to have a great practice. One problem is that technology is moving fast, and the industry has been slow to evolve and depending on the size of the organization. Many times, organizations cannot keep up due to antiquated outdated release cycle processes. Performance has many times been associated with just testing, not including any deep performance analysis as part of a release. For example, we remember not that long ago being considered for a remote position at one of the largest medical companies in the world. It looked like a great job leading over a large performance organization across their enterprise. They told me they were having challenges because teams were testing and doing whatever they wanted with respect to performance; it was impacting enrollments and application stability. In this group, as an IT team, they owned the practice and infrastructure but felt that they had no way to drive a good practice because they were a horizontal team with limited resources. They were an organization of over hundred thousand people and had not yet evolved beyond performance testing. I told them even if they were doing just testing and were not yet mature in their processes, they needed to understand if SLO were being met for a product. More importantly, the first step was to determine if testing itself was even being done correctly.

Not sure how many times that we have seen performance testing performed in a lower environment where it was very different from production with respect to data sizing and configuration. This makes the testing completely invalid. If you do not take the time to establish a good performance process in an organization, there is no use of even testing. Why? Because you would be generating more cost from bad testing and chasing down red herrings where problems did not really exist rather than truly providing value.

When it comes to performance roles, to this day, still a great deal of confusion exists around responsibilities. Just look at *Wikipedia* or other sources of reference and you will see definitions that say that engineering platforms for performance is something that organizations do around testing software before it is published. It can be very confusing and your take on roles and responsibilities in many ways may translate to your background or be a derivation of existing processes put in place. Anytime we see something on the web we must question it.

The industry is confused by the literature and the source of that literature in many instances. Just like the news can be biased towards a political party or towards a biased understanding to drive behavior or votes, it is important to rely on credible sources when it comes to a good practice. Credible sources are generally the people on the ground engineering platforms for good performance versus blogs written by vendors of load testing products or someone having an affinity towards a given product. Let us be honest, achieving great performance is in no way related to a product's ability such as load generators or observability tools, it is a practice consisting of processes to assure good platform performance.

Check in the box testing

Some organizations still think in terms of check in the box testing looking for those performance outliers which may impact a release and nothing else. What do we mean by check in the box? It means that performance testing is a last-minute consideration with a squeezed window of time and is only a quick high level test that happens before the product goes out the door without any deep analysis around performance bottlenecks. The goal of these tests is just to make sure that the platform reaches the needed processing rates around capacity and it is a popular approach for manual testing in release cycles to speed up delivery. This is especially true in performance practices that evolve from groups not completely understanding good performance processes or groups that do not have the needed performance automation or quality gating in place around releases.

People usually embrace what is familiar and think about existing processes versus forming new ones. For example, a large organization who needed a great rewards platform for their product, wanted a distributed platform with web services written in Java to be the solution to assure they had some flexibility for working with other groups or businesses for passing information back and forth. Due to timelines and resource availability, many of the architects put on this product were from a data engineering group, they were database designers and architects that design architecture/processes for data warehousing and operational data stores. The database architects, to their credit, were able to design a fast and working large, distributed platform to provide customers rewards on credit card transactions. No doubt about it, they quickly rose to the challenge and delivered. But ultimately, due to their skillset what kind of product architecture do you think was created? As you can probably guess, all the business logic and most of the application was written in the database itself with 80,000-line stored procedures and small snippets of Java application code calling the large database procedures.

In other words, something was created that was very hard to maintain or have any flexibility with to adapt to change that snowballed into something slower with time as those procedures got bigger and continually needed to be replicated. Just like this was a bad architecture because it was not designed by Java architects versus database experts, a bad performance practice will occur if you have the wrong group derive it. People that are writing slow software consistently for example are probably not who you want to choose to establish your first practice.

The practice will only evolve well if the practice is created by individuals or teams who truly understand all dynamics of the IT organization and understands the proper separation of concerns when it comes to processes. Architects that oversee multiple groups and understand group dynamics at a high level or site reliability engineering groups are generally the best candidates if establishing something brand new. Outside consulting firms can also be good candidates if they have the right expertise and take the time to fully understand organizational dynamics.

However, engineering processes around improving performance is not just testing.

for an IT platform. Just like capacity planning is its own domain and understanding it is very important to understand how to do capacity management and how it correlates to performance and cloud quota limits which dictate the hard limit around cloud billing for transactions. Performance teams provide consultation to development and operational teams around building performance into products and help identify root cause issues/resolutions to performance problems with existing platforms. Testing and capacity planning metrics are just something professionals use to help them understand the platform dynamics. For example, metrics such as **transactions per second** (**TPS**). TPS generally refers to a business transaction which is one or more requests grouped. As response times increase capacity, TPS typically decreases so there is always an inverse relationship between TPS and response time. As response time goes down, TPS goes up and as response time gets slower, TPS goes down. TPS is a great measure of throughput or processing rate for a given transaction to understand its scale.

The performance practice is in reality a holistic approach. The most basic definition means to simply understand how to design/architect a platform for the greatest performance and that requires being able to understand the full distributed technology stack. The practitioner should be able to isolate root causes associated with performance when issues do arise. It means understanding architecture, software design, code, storage, anti-patterns, operations, the network, and overall best practices to build a performant platform and to understand the platform so completely that you understand the dynamics associated with performance. It also means understanding what if scenarios. Understanding if the platform dynamics change while in many cases understanding how to redesign existing platforms to help platforms achieve a great customer experience.

Usually, the best performance architects and engineers are people with skillsets that understand how to build distributed applications, understand operations/network, understand complex software and performance practices. When engineering a platform for performance, it means understanding the platform both from a hardware, software, and network perspective. What to look for when hiring someone into a role of managing performance for a platform? Someone that is a great architect than can program, understands operations, and understands the performance practice we may select. However, you cannot only test software unless you have evolved yourself to a higher level of understanding; you need to understand that both the software and hardware dynamics that are needed to be a great performance engineer. The goal should always be to identify root cause issues around slowness and tell teams how to make software or hardware faster, not just generate a report and say it is slow and not passing the test and toss the problem over the wall to the other teams to try to figure out what is really going on. It is important to have observability on the platform, profiling capabilities and enough understanding to determine root cause and recommend solutions.

Roles and responsibilities

Engineering platforms for great performance means doing code reviews, design reviews, architecture reviews, testing and infrastructure/data

software to work together to achieve the best performance. It means stepping left early in the development cycle to bake in good performance practices sooner rather than later. Capacity planning to understand how the platform needs to scale based on an expected traffic pattern comes into the picture after the platform is optimized.

Optimize first mindset

It is important to have a good **performance process** in the organization. From a performance and scale perspective, the goal is to always optimize the software first, understand the pipeline, complete any intelligent testing/measurement activities, and understand capacity drivers. For example, one of the worst behavioral patterns that we see in general is that platform optimization becomes reactive based on testing in many organizations. Hence, an organization has their performance testing team run a test, find the bottleneck, run a test, find the next bottleneck, run a test and find the third bottleneck, etc. Consider all the configurations associated with both software and hardware every step of the way and consider how this reactive approach is very counterproductive. It can be an extremely slow process, especially if other team dependencies exist.

When the performance engineer or tester knows that 100 requests need to come in simultaneously to a reverse proxy as an example, throttles (thread pools) should be set ahead of time to support the requirement in that reverse proxy. There is a great deal of pre-testing activities that can really save time. As mentioned before, we do not just want to tune the bottlenecks, we want to optimize the platform out of the gate. If a multithreaded process is in place and the software configures via an external file how many processes it will run in parallel or if a web service has configuration around connections, threads, queues, etc. and we know what the load needs to be at the end state, those settings should always be preconfigured. We should not run a test to find this and then act, we should perform the due diligence and make this a part of the pre-testing process. If a performance expert knows that 20 requests translate to 5 database queries and that 100 requests simultaneously is the requirement, then the database pooling should be set to 25 or a value less or more based on accepted latency around the pooling. If a performance engineer analyzes web page performance and they know that the page is not doing any caching, is loading a 10 mbyte image, and has 300 requests across the wire, should they test it anyway? The answer is no! Understand the environments, software configuration, do focused testing any anything not understood in a limited fashion and pre-tune the software and hardware before any full capacity testing is done. If you take this approach, you will thank me later, it will save weeks in testing.

Always tune the infrastructure and fix the known bottlenecks to support the requirement ahead of time and always make sure the platform's infrastructure is optimized for what needs to be optimized ahead of time. It will save countless hours in optimization time and eliminate the mentality around running 200 performance tests just to optimize an organization's platform. It should not be reactive; a performance engineer should understand the profiling dynamics of the application and make sure that the platform is

configured ahead of time to support it before the first test. Always be proactive and do the necessary performance analysis before even doing one load test and reduce the testing interactions required. Put what is needed to accept the required traffic into configuration ahead of time and the same goes for known **software bottlenecks**. Then run the test and you will find yourself doing very finite changes vs. large changes and a much smaller time frame. Mathematically performance engineers can extrapolate the resources they need doing simple analysis when that is an option.

Understand the customer experience:

Another poor pattern I see around testing is that performance testers in general are that many times the frontend performance is not considered or tested under load, which is half of the equation around performance with applications leveraging new modern frontend frameworks like Angular, React or Vue as an example. Why? For decades performance testing just meant checking the capacity of the platform and not really the user experience. Why do we care about this? Will the customer be frustrated and leave? Will they get kicked out due to browser timeouts for long running requests or backed up browser threads? Most modern APM tools now have incorporated a frustration index where they can set alerts when certain thresholds are met for users going down paths in an application or if the user bounces (leaves or gets kicked out of the platform). Testing the backend with load generation tools like JMeter, which is a free tool that does not measure JavaScript and the frontend rendering, without combining it with Selenium for the frontend only gives you half of the equation around performance. You can also use an application like LoadRunner frontend tools like TruClient to understand what the customer would really experience. Both the front and backend performance must be measured for a platform that contains a web frontend. The frontend can also be measured using tools like Performance Insights in *Chrome Developer tools* or in other browsers while the platform is at peak load for these backend tools as a manual way to take these measurements. Many tools to do this these days.

Engineering versus performance testing

What does it mean when you say that you are a performance engineer working in IT? Does that mean that your job is to do performance testing on IT platforms? This is one key point that we want to make early on in this book. Performance testing is a practice that has been around for a very long time in IT. It involves writing scripts to test a web product or service in most contexts, though it may be a database or key component of an architecture. It may even mean testing hardware components specifically such disk drives or CPU processing times for a server. From a holistic understanding around testing as part of most IT shows, at a very high level, it means understanding the incoming remote/local traffic pipeline to a component or application, establishing a workload model (how traffic is routed across requests) and generating load and analyzing the results. This description is grossly an oversimplification around this role, but this is a performance tester's function when it comes to new platform delivery. It is a specialized talent that requires a lot of expertise to

be good at it. Performance testers will script and run the test as part of a software delivery process before a release. The testing in most situations is a manual testing process; they use software to generate load to a target platform and then record the processing rates, response times, error rates, etc. and publish a report to stakeholders.

Engineers focusing on performance do much more than testing, though testing is also a part of what they do. Good performance processes involve multiple teams who are responsible for all aspects of a platform's performance. This includes processes far left early in the design/development process to right in the production environment. Engineers focusing on performance fully understand automation around performance measurement and quality gating across environments. These folks understand the SLO, the architecture, code bottlenecks, and much of the software design. The role these engineers play means understanding performance measurements throughout the lifecycle of the product. They work with teams to identify root causes and help optimize bottlenecks.

Engineering platforms for performance still includes testing, but it also involves optimization of software and infrastructure to assure great platform performance and working through root cause bottlenecks for the platform. The process involves multiple IT teams including development, site reliability engineering, performance testing, quality engineering, and most importantly the business stakeholders who fully understand the incoming customer pipeline and response time requirements as well as availability.

Performance tester

What does a performance tester do as part of their job?

- Writes scripts in a load generation tool
- Defines the workload model which is the expected traffic pattern that will exist in production
- Runs performance tests in an environment similar to production
- Compares results to SLO defined as part of the platform requirements. In other words, what performance target to we need to hit

Presents or writes reporting to showcase the requirements are being met for each release and identifies where requirements are not being met from the test.

Let me go deeper here and give you some comparisons for context. For an already existing IT platform vs. a new platform the processes are similar but still a little different. For an existing platform, more accurate information is understood around existing customers using the platform in production which means we can be very accurate in how the performance tester can test. New systems require expected load and traffic pattern requirements to be given to them by the business folks defining the requirement. The requirements defined in my experience are very much more hopeful than realistic in many cases because it is in anticipation of making sales and onboarding customers. For an existing platform, it is an entirely different story. In this case, a performance

load levels using observability tooling, and determine how many requests are going into an application, service or component. They will derive a realistic workload model of the existing traffic. The workload model defines the percentage of traffic going to each user journey or individual component in the application. Then they write a script with test data that closely matches the data distribution in production and run tests at production load to make sure that it is meaning performance service level objectives in a separate environment that matches production.

Performance testers need observability to profile an application

For a performance tester to be able to do a great job, good platform observability is required because they need to be able to replicate the performance dynamics seen in production. They need observability both in the production environment and the environment that they are testing in for comparison to see where deltas between the environments exist. They can then profile the behavior of each environment and quickly identify differences. It is very important for a performance tester to know if any difference exist between the environments so that they can achieve a valid test. For example, a tester can be using an environment with 2 servers in the lower environment with four core processors of a particular processor family, bus frequency, etc. In production, they may be using 6 servers with a different number of processors, perhaps one has 64 bit registers and another has 32. If the server count, memory, processor type, bus frequencies, memory, etc. are different between environments then this is a situation that needs to be considered and accounted for in how much load is generated and how to interpret the results. If software or hardware configurations are different between environments or the database resource dynamics are different, perhaps it has different record counts than production or a different amount of CPU cores, then a performance tester needs to consider this so that they can mathematically extrapolate or measure the differences through profiling which we will go into deeper later in the book.

More than anything else, understanding the environmental differences is the key to having a valid test. Running tests on dissimilar environments without understanding the differences is a recipe for disaster when it comes to performance testing. Unfortunately, it has happened far too many times where a load test is run in a lower environment that does not match production, bottlenecks are seen and hundreds of man hours are spent trying to fix something that was not a real problem to begin with, it was just an invalid test. If you are new to performance testing, you may ask why would an organization have a lower environment not matching the production environment 100% with respect to hardware and configuration? The reason is cost. For very large platforms that have hundreds of servers, many times it becomes too costly to have identical server counts. In these situations, you want to use a factor multiplier and identical processor, memory, database sizing, etc. Hence, if you have 500 servers in production you can still have 100 servers in the lower environment configured identically and test at $1/5^{th}$ of the load and take into

account uncertainties and balancing latencies/context switching between different request routes though the platform. This becomes much easier to do if everything else is the same, balancer configurations, horsepower, reverse proxy configurations, JVM configurations, same CPU and memory, etc. Understanding these differences and minimizing variables as much as possible is the key to having a valid test. Without a valid test, what is the point of testing right?

Different performance test types

When we talk about the different types of testing that can happen there are some standard tests that occur for most platforms. These tests assure that the platform performance dynamics are understood, whether this be a situation of a traffic surge or a memory leak that is occurring over time. Different testing types involve, not only capacity limits, but also behaviors resulting in identifying poor resiliency and negative performance scenarios. Understanding how the user experience varies when the platform is under different types of duress or edge case behaviors helps engineers determine how to better optimize the platform.

Spike testing

Testing quick ramp up of traffic representing the worst case expected ramp up event. As an example, let us say an application is designed to allow users to purchase concert tickets and as a promotion an email is sent out to a large group of consumers targeted for the marketing campaign. The email goes out to 20,000 consumers at the exact same time with the expectation that 10 percent of those consumers will click on the link to purchase the tickets. 2000 people all at once click the link to access the website resulting in excessive load onto the platform. Let us say the email campaign goes out to participants as part of a platform's batch processing at 3:00am in the morning. When consumers log into their computers or phones in the morning, they see the link. At 8:00am, a large amount of requests start coming into the platform. These flood the platform yet again with traffic, but with all of that traffic coming in all at once in the morning does the application get slower? Do customers still have the same experience as if the additional traffic was not taking place? Can the system absorb the traffic without getting slower or dropping requests and failing due to long running threads that are timing out in the backend? Are SLO around response times and availability for the platform still being met? What types of error rates are being seen? What is the delta between nominal load onto the platform at 8:00am and the inflow of new requests due to the campaign? This ramp up delta is the load that a performance tester would put onto the platform to understand its behavior.

Soak testing

This is at times also called **endurance testing**. Performance testers will generate at load at 50 percent or sometimes up to 80% of the traffic and test the platform for 6 to 24 hours to understand the platform behavior. What they are looking

times, increased error rates, and impacts to CPU and memory at a high level. At a deeper level, they are looking for memory leaks (most importantly) and reviewing connection pools backing up or requests queuing into the application or service.

As an example, let us say a platform runs with 20 users on the platform at the same time with nominal load 24 hours a day. If the platform starts out consuming 20% of the CPU and 50% of the memory with response times of 2 seconds and error rates of 0.1%. At the end of the day are customers experiencing the same response times, error rates? Has CPU utilization increased, or has it remained constant? How about memory? Did memory usage increase over time? More importantly as traffic spiked and leveled off, did memory reduce to its original value as expected? Soak testing that performance testers run determines the resiliency of the platform over time beyond determining the customer experience. What is generally discovered is that memory leaks over time due to bad code. When we say bad code, what does that really mean? It means that references to objects are held onto in memory and the JVM's garbage collector could not clean up the memory. It means the object in memory was never dereferenced or set to null in the code. For Java based platforms, the JVM garbage collector is used to clean them up and free up memory. One can also clean up objects programmatically in many languages as well. But memory leaks happen when memory is not freed up over time. When running a soak test, performance testers can catch this and help keep the platform from crashing or degrading the customer's experience over time. It reduces the **mean time between failures** (**MTBF**) which is the average time between system breakdowns. If I have a slow memory leak that eventually maxes out the available resources on the platform every 3 weeks, this is a problem because it reduces the availability to the customer.

Performance load testing

Though sometimes called by different names, when someone does performance load testing, it means they are testing the software that matches the production load level and verifying the response times, processing rates, and error rates all meet SLO defined for the platform. Let me give you an example. If a platform exists in production already and that platform peaks at 10 requests coming into it concurrently, then you test at 10 requests coming into the platform in the lower environment and you measure the response time, error rates, and processing rate. These should be in line with the defined SLO for the platform. So, a basic performance test is what an organization will typically do for each release. The development team writes new software, performance testers then script the new functionality and augments the workload model being used as part of the test based on expected usage of the new functionality. They then run a performance test at the production load level to make sure that the platform is still meeting the SLO and that the new functionality created did not degrade the customer's user experience. This has been done wrong by many organizations. Some organizations will run at 2 times and 4 times or larger than what production requires because they feel it gives them a level of comfort and overhead but not setup SLO or profile behavior of the application at that load level. Unless SLOs are defined at that load level (these are generally established for a business

purpose and should be included around requirements) and differences between the lower environment and production are understood at that load level, one cannot reliably determine if the SLO is being met or if the SLO even makes sense. That is what capacity testing is for after all discussed next which will determine where the thresholds are to resistance around scale.

A basic performance test should be measuring what is required to go out for a release at production load levels at that given point in time. Combining load and capacity testing together to save time is not a good idea. What you generally want to do is perform performance load testing for every release and capacity testing less frequently (monthly or quarterly) to see where the upper limits are. It is fine to add 10% or 15% above production load level to minimize uncertainties around test deltas. For example, if we use a load generation tool like JMeter and we run a load test, we will get one set of results. If we run the exact same test again, we will see some level of delta in the results. That standard deviation in response time and other metrics may be 5 or 10% of what was achieved. Why does this happen? Load generation tools typically have their own infrastructure setup, they spin up threads, go across common firewalls, common network switches, common balancers, and common reverse proxies. Common infrastructure and how long connections are active across requests impacts the results of the test and consistency of the tests. Beyond this log sizes, disk latencies, etc. all play a factor beyond runtime around the software itself. Even JMeter as a load tool will contribute 200ms to a request from a response time deviation perspective on its own. Important to understand the deviation or put a factor of 10-15% on top of the test or the level of deviation being seen across at least three test runs.

Capacity testing

We have seen this done in so many ways. This test, beyond all others, is where most people not familiar with a good performance practice in their organization fail. First let me say that the most confusing part of capacity testing is that most people do not fully understand it. Sometimes people in IT will call this Breakpoint testing. They feel that you run the platform until it breaks and until hardware falls over and no longer functions. Let us understand what capacity testing really is. It is when you continually ramp up more load on a platform to the point where SLO are not being met, period. The capacity of your platform is only to the point where the customer can really use the system, not where the platform is unreliable or when features of the application are hit or miss on whether they will work or not. A platform with wild swings in response time above the SLO is also not something you care about for your customers. In all platforms, at some point, there will start to be resistance to the traffic being sent to it and as a result response times increase, perhaps timeouts start occurring at different layers of the infrastructure and error rates increase as well. The good news is that when it comes to capacity testing, you do not need to start breaking things and start hanging up threads or requiring your operations staff to reboot servers or restart services in the cloud. It is a humungous undertaking for on premise systems and even though more convenient for cloud platforms, it is still a large

understanding for a large platform. For capacity testing, a good performance tester will always test to the load level required for the incoming pipeline expected at 50% capacity of your platform as a practice and then determine if the SLO are being met. So, if the requirement for your platform is to handle 200 users concurrently with response times of 2 seconds and error rates of 0.01 percent for requests, then as soon as those users exceed those values, you stop the test. That is your capacity and you have met the requirement. Now some people will want to run the load higher anyway just to see what happens out of morbid curiosity, but as a practice you want to configure your platform to support 2 times of what is expected and test the SLO to your expected incoming traffic pipeline. So, you test to 50% of your configured capacity and make sure that SLOs are not exceeded.

Looking to understand the integral roles associated building an IT platform with great performance engineering? Below is comprehensive information to shed light on each role within an IT organization, and precisely how they contribute towards the performance of a platform. We will delve into details, insightfully explaining each function, and how they jointly create an optimally functioning IT platform. Let us now get right into it.

Product architect/Business analyst

Depending on the organizational structure, there is always someone defining the requirements and the working with regards to how the platform will work. They understand what is being delivered and accordingly obtain information around the incoming pipeline to the feature as far as customer traffic magnitudes. From a performance perspective, they evangelize the importance of platform performance to the business and always consider performance as a feature or SLO that needs to be embedded for each requirement. For example, if the product is an e-commerce product that is used to order clothing, how long should it take to order that clothing from a user journey perspective. User journeys refer to the path through the application to fulfill a task.

Platform architect

A lot of different platform architectures exist in the industry, some are distributed, and others are monolithic in nature. But all of them interact a little differently with respect to interoperability between components and proper separation of concerns; having the component in the right architecture layer in the right domain being one example. Many have different communication protocols, architecture layers, some are Polyglot and self-contained with smaller amounts of dependencies and others are larger with common code and dependencies. Some are **service oriented architectures** (**SOA**), some are microservice architectures, some event driven while others are more synchronous and direct in nature. Platform architects generally think about flexibility and maintenance around a platform in the future and resiliency around component interaction. They want to design something that can easily adapt to changing requirements to minimize the impact to products allowing an organization to be very flexible saving future development costs. They generally follow good enterprise integration design patterns to assure the flexibility and

interoperability with other domains in the enterprise and their architectures. Architects are responsible for architecting responsive platforms that meet the needed performance requirements. In other words, architectural decisions need to be made to keep the platform fast and platforms need to be architected with that decision in mind to meet the defined requirements. Generally going async vs. synchronous and parallelization techniques are regularly fostered in architecture overall but understanding how encryption, fine grained processing vs. coarse grained, keep alive and pooling impacts scale is important in what is setup. They are architecting a flexible platform that performs and balancing performance and flexibility is a very difficult thing to do, it is an art within itself. They heavily rely on the performance team to understand the platform's performance dynamics over time.

Performance architect

Involved in the entire technology stack evolution around product creation with a focus on performance considerations from architecture, infrastructure, design, coding, capacity planning through to testing. They typically own the performance practice and processes around performance and are the glue that ties them together. These folks have a high level of expertise and understand all the bottlenecks around performance for a product from both a software and hardware perspective. They continue to perform focused testing, analysis, profile and review code, and continually look for performance anti-patterns that are preventing a platform from reaching its full potential with respect to performance. They do proof of concepts early in the product creation process to provide guidance around how to make things faster to save costs farther down the road and are always kept in the loop around capacity and performance requirements.

Performance tester

Execute performance testing typically in the development cycle for platform releases. Create scripts based on a workload model around traffic patterns. Test changes related to code change specifics and run focused tests or continually test to the top 10 flows and understand the impact to capacity and performance for a release.

Software engineer

Being a software engineer means much more than simply writing bug-free code these days. As highly distributed apps become more complex, developers need to ensure their systems are as user-friendly, fast, secure, and scalable as possible. When developers do not do this, revenue is lost, and bounce rates increase significantly. One simple example is when developers make the simple and common mistake of using exceptions in code to throw up the call stack to direct program flow, the Just in Time compiler used with Java cannot optimize that, thus causing the application to be less than performant. In fact, if you do poor error handling and generate a stack trace you encounter the equivalent of 100 lines of code executing. The call up the stack unoptimized and the stack stream will cause this latency. Good coding practices matter when it comes to performance and understanding

the good patterns and anti-patterns around performance is important when it comes to developing great code.

Site reliability engineer

Site reliability engineering became popular when software started moving to the cloud. These folks are the new updated operations folks who are experts in cloud automation and monitoring of the platform. They setup alerting and manage/own the environment to make sure it is always available and performing. It is important to consider them both build and run experts, people that can build great infrastructure and monitor runtime environments. When the software or hardware is not performing, they have thresholds setup around alerting to notify the appropriate folks and take the needed action to get the platform running fast and reliably again. From a performance perspective, these folks' setup the dashboards and closely watch performance for anomalies and alert when they exist. In many cases, they take on the old capacity planning roles for the cloud.

Quality engineer

Quality engineers are responsible for ensuring the platform is functional before performance testers are engaged. They are also automation experts around testing against both functional requirements and performance in many cases. In the industry, most of these folks are experts in Selenium or other tools like Playwright, Cypress, etc. and when it comes to performance, tools like these have great performance measurement capabilities around frontend measurement and when combined with other load generators, it can be easy to get the full picture around performance in an automated fashion. Between Selenium, load tools like JMeter, Gatling, LoadRunner and tools like **Java Microbenchmark Harness** (**JMH**), these folks can do a lot of during integration testing and upper environment end to end testing. You see a lot of overlap between quality engineering and performance groups and responsibilities which is why you see cross skills here on resumes. We see a lot of quality engineers with performance experience on their resume as well.

Capacity planner/engineer

Capacity planners were dedicated folks that were experts in platform performance for on-premises platforms. They monitored servers and understood when CPU and memory or other thresholds were being breached and would help with acquiring more servers when needed. They understand the pipeline and ideally the model around behavior to determine when more capacity was warranted and run regular monthly capacity reports. These folks are experts in Linux utilities like SAR, perf-stat, Dtrace, IOStat, etc. and others which were critical in understanding an on-premise server's capacity. With the new cloud environments, it has become all about quotas which is just the limit of how much horsepower is allowed to prevent the bills from getting too high. We can see capacity planning falling into the site reliability engineering areas or the performance testing team areas most commonly vs. having dedicated capacity planners. Not to say that

larger organizations are not working in a hybrid fashion with both cloud and on-premise architectures requiring a dedicated capacity planning group.

Performance tools

In this section, we will go through some performance tools:

- **Software code analysis tools:** These are tools that analyze the frontend and backend code looking for poorly written software patterns with respect to performance. Popular tools are *Chrome LightSpeed*, *Page Speed Insight*s, *GTMetrix*, *PMD* and others at the time of this book's writing, but the need for such tools is timeless. For a good performance practices, it is important to run code analysis to understand where the code is not written for the best performance or when proper design patterns are not followed. For example, is the data compressed going across the wire between servers. Is JavaScript and CSS minified, have images been optimized, is **time to live** (**TTL**) caching (caching for spans of time implemented across layers of the architecture—we will talk more about TTL caching later to speed up serving requests, etc.), and have non-blocking scripting patterns been put into place with async logic where it can be. Have **Doman Name System** (**DNS**) been pre-fetched or SSL negotiations for thread party dependencies while the browser is idle to speed up page load times, etc. Has large JavaScript library modules been split into smaller modules to download and has only the code used for a framework been leveraged; the list of optimization opportunities goes on and on. These tools help quickly catch poor coding patterns early in the process.
- **Code runtime profiling tools:** Runtime profilers are essential, and most cloud providers have a profiler to use, such as the *Google Cloud Profiler* created by *Google*. Other popular profilers at the time of this writing are *JProfiler* and the *Java Flight Recorder* which is built into the JVM when it comes to Java. APM tools generally will leverage these other tools under the covers and give some nice screen management/data aggregation capabilities to drill into issues.
- **Load testing tools:** There are a lot of tools for generating load in performance testing. All of them are pretty good at doing this, it is not complex. How much standard deviation from one test or one measurement to the next? Popular tools like *JMeter* will have 200ms standard response time deviation on average, Micro benchmarking tools like JMH combined with multi-threading in Java with even priorities which see only a few milliseconds with much less overhead and horsepower required. When it comes to load generation, you want something fast, that requires minimum CPU and overhead. The most important thing to consider when picking a tool for today's applications running on cloud platforms is the load generators in the cloud as far as where they reside and how repeatable they are. For example, do you have both on-premise and cloud platforms and do you need the ability to generate load both on-premise and the cloud? Does the tool support

testing from different geographical locations at different bandwidths? Does the tool support different browsers in a real-world fashion with the same thread and rendering dynamics? For web applications, does the tool test the frontend or just the backend by default? The most popular tools at the time of this writing are *LoadRunner* with *TruClient* which supports both backend and frontend website testing. For many years this has been the most comprehensive testing solution out there. Other popular tools are JMeter, Gatling, JMH with Java web client threading, Neoload, Locust, Grinder, k6, and many load generation tools are built into observability products like APM tools. There are also many performance load generators that come with web servers like Apache and others. These products are generally command line only and very easy to use to generate load but have limited feature sets.

- **Observability tools:** Observability tools gather telemetric data metrics for an IT platform. These are very important when it comes to understanding what is truly happening on an IT platform under different load traffic patterns. We cannot emphasize enough how important it is to have good observability in place in lower environments before ever doing any testing; it will save countless weeks of work. We have seen organizations wait until the end of product development to put observability in place thinking it is something only for production. This is a terrible mistake. Observability tools in lower environments paired with profiling and load testing tools quickly isolate root cause to performance issues without guessing. Popular observability tools at the time of this writing are *OpenTelemetry*, *Dynatrace*, *Datadog*, *New Relic*, application dynamics. *Splunk*, *Signal FX*, *LogStash* with *ElasticSearch*. But there are a lot of tools that do this and many also include insights to the database, just pick one that meets the requirements that are needed for the implementation. If picking a tool for a large enterprise, it is generally safe to pick the one in the Gartner magic quadrant to minimize financial risk. The cloud observability tools built into cloud providers today generally are more limited using Java JMX protocols or tooling interfaces into runtime environments gathering metrics along with built in log parsing. But they do have decent database analysis tools for their own DB solutions in many cases. Cloud monitoring has much less AI capability than many of the other commercial solutions with predictive modeling. But they are getting much better every year and have the potential of eventually replacing all the other observability tools. Where most of the third party APM and Log analysis tools have been excelling around observability has been in AI when it comes to identifying issues on the platform and root causes quickly. When it comes to production, especially finding root cause or frustration from customers quickly is of huge importance and not just being able to do that but being able to do that in a few seconds or before the customer is hugely beneficial. One popular tool that many cloud providers have embraced is Prometheus which they embed with their products for observability and even more comprehensive horizontal scaling around load patterns. This tool is popular because it is much lighter weight than

the metrics. Many tools have very large agent sizes that they must embed with the runtime environment. If you are writing source code that is only 500k in size but you have to deploy a 250 megabyte agent to monitor it, it can become very costly. Prometheus includes time series data storage and dashboards or leveraged databases like InfluxDB (which is popular to use with capturing data with some of the load generation products over time as well). Generally, data from Prometheus or other tools can be pushed to third party dashboarding products like Grafana which has a built-in query language that is popular to use. The bottom line is that organization's need good observability across their technology stack in all environments regardless of the tool and to be able to store that data over a period of time for short term analysis. Storing data for 3 weeks to 90 days is popular.

Conclusion

We discussed various topics in the first chapter to provide a good introduction to engineering IT platforms for good performance. Now you should be able to begin to understand the performance challenges and concepts involved when creating new IT platforms. You will fully understand the different roles and responsibilities involved in engineering fast platforms and some performance best practices required for an organization to be successful.

In the next chapter, we are going to cover some exciting topics that will be all about building a great performance practice in your organization. It can take many years to understand how to create a good practice, but we will simplify it for you and help you be successful. We will be going much deeper and build upon what you have already learned but discuss performance considerations more in depth and how they fit into the overall picture of IT platform development.

Building a good performance culture in IT can be a challenging endeavor for any organization. We will go over why it is so important to be successful. Then, we will discuss different development methodologies and where performance falls into the picture for each methodology. Approaches to performance measurement will be discussed along with some small code examples to help you fully understand how to implement performance measurement. The project iron triangle will be introduced and summarized for you understand the delivery dynamics around project management in the industry and how these dynamics presented must be fully balanced to assure great performance. DevOps concepts will be reviewed, along with the advantages of adopting a good performance mindset; always considering performance first in the life cycle for product development. We will briefly touch on the Agile Manifesto and where different processes excel and fail so that you will take these understandings and know how to build great IT platforms that perform well. We will also talk more about software design considerations and how to stay ahead of the competition.

Chapter 2
Building a Performance Practice

Introduction

This chapter of the book will introduce you to creating a good performance practice for your organization. It will discuss the creation of a performance culture, the different processes and methodologies around software development, and, most importantly, where performance plays a key role. **Development operations** (**DevOps**) will also be discussed, along with continuous testing practices. Remember that we are building upon the theme in chapter one and going to the next step. A performance practice is a systematic approach that ensures that software meets its intended performance criteria. It is much more than just running performance tests that you may be familiar with like JMeter or LoadRunner tests where a workload model is defined around traffic patterns and generating load. The performance practice nomenclature used in this book refers to a comprehensive strategy that integrates performance considerations throughout the software development lifecycle. This allows us to not only identify potential bottlenecks or constraints early but also design and implement solutions that enhance the software's scalability, resilience and speed.

Structure

This chapter covers the following topics:

- Performance engineering in software development

- The project iron triangle
- Software development methodologies
- Technological innovation
- Total quality management
- IT performance cycle delivery process
- Site reliability engineering
- Performance first product development process
- Defining a workload model

Objectives

By the end of this chapter, you will be able to understand the different approaches to product delivery and, how performance plays a role in the product creation and why it matters. Organizational culture is discussed along with Agile and DevOps concepts around product delivery. The reader will understand some of the project challenges around the different methodologies and where performance plays an important role.

Performance engineering in software development

As an application or software developer, one of the critical roles to play is providing an exceptional user experience, assuring that the product meets or exceeds the expectations of the customer. This responsibility goes beyond just creating a good user interface or IT platform that is functional and works as it is expected to. It also ensures that the software optimally gives the customer a great experience. This helps build the organization's brand and the customer's perception. This is where the performance engineering process becomes important.

Consider a website that you frequently visit. If it consistently takes too long to load, you may start considering other alternatives as we talked about some in *Chapter 1, Introduction to Performance Engineering*. Remember that performance engineering in the software development lifecycle revolves around designing, building, and testing software applications to fulfill specific performance requirements. It is about ensuring that a system can handle anticipated user loads without compromising the user experience. That includes minimizing response time to user actions, managing system resources efficiently, and addressing potential bottlenecks that could slow down the performance. But it is more than just speeding up an application in the end. It is about integrating performance considerations throughout the development cycle, which is the only way to achieve great performance.

Over the years, we have witnessed that many organizations try to save performance to the end and try to magically refactor what is needed at the end. This is derived from some early waterfall processes where performance testing was the last step in the process before a product went out the door. This path leads to poor design patterns and large costs due to a lot of rework. Different software development methodologies dictate how you incorporate performance engineering practices into your software development process.

In this chapter, we will take a brief look at some of the most popular methodologies. But before we do that, we need to consider that to have great performance, regardless of the process used, we need to establish a good performance culture in the organization. The most important aspects of building performance into the product lifecycle involve the following considerations:

Building performance culture within development teams

If you really want fast software, great revenue, brand recognition through an award-winning product, then building a performance culture is essential. This can be a difficult thing to do, but do not give up. Building a performance culture within development teams is an integral part of any successful software development project. This means evangelizing performance and a mindset where each team member completely understands the importance of high-grade performance outcomes. The team then works collaboratively to achieve this common goal. Here is how you can foster a performance-driven culture within your team:

Education

Many software architects, software developers, and businesspeople do not really understand performance to the point of averting performance challenges upfront before they impact the customer. From a businessperson's perspective, stakeholders in this area would just want it fast. But this is where the education comes in. The first step to building a performance culture is to ensure that everyone on the team, including the business driving the requirements, understands the importance of performance to the user experience and the overall success of the application. Holding regular educational sessions and making it a part of the process around onboarding new resources is important. If you always continually highlight the need for a performance-first approach, it can go a long way in cultivating this important mindset that evolves the culture. This can be evangelized from the bottom for education and understanding but ultimately it needs to be driven from the top as something being important. If it is not important today in an organization, you need to start evangelizing why it is important to the business and use revenue as the driver. Once revenue comes into the picture and the understanding around potential loss of revenue, it can become important fast. Always remember when presenting something that is needed to drive change, leverage is the key. People need to understand the benefit,

not just that it is a good idea and revenue impacts to an organization or the bottom line is a good leverage approach to driving change. There needs to be a benefit as to why the change is needed and why we care about performance and it cannot be personal or what we personally like, it has to make a difference to selling the product, revenue, or something that truly helps the business and is measurable.

Establishing clear performance goals

Having defined and measurable performance objectives will help the team to know exactly what is expected of them. These goals should align with the organization's vision, and the product's desired outcome. Later in this chapter, we will talk about the different steps and phases to include performance measurement in the regular development cycle, which can help ensure that the team is making consistent progress towards the defined goals. Performance goals ensure everyone understands what success looks like. By setting **specific, measurable, achievable, relevant, and time-bound** (**SMART**) goals, we can promote continuous improvement and accountability for the team.

Sometimes, in less mature organizations, performance is overlooked in product development to the peril of an organization. In Agile processes, clear performance goals align with the core principles of regular reflection and adjustment but it matters in all processes being used for the software development lifecycle. They allow teams to track progress, evaluate outcomes, and make necessary adjustments in real-time to achieve great performance and thus a good user experience. This iterative approach not only keeps the team focused and motivated but also helps in identifying potential issues early, leading to more robust and efficient solutions.

Moreover, clear performance goals enable better resource allocation and prioritization. When everyone is clear on what needs to be achieved for performance, it is easier to allocate team capacity towards tasks and ensure that critical objectives are met on time. This clarity helps in delivering high-quality products that meet or exceed stakeholder expectations.

Performance reviews

Frequent reviews and feedback sessions can contribute to consistent progress and improvement. When team members see their work's impact from the performance perspective, it can motivate them to improve further. Incorporating performance reviews during the code review and design stages is critical in the creation of resilient software solutions. This includes reviewing design patterns incorporated to make sure the best high performance design patterns are followed. While these reviews traditionally focus on code quality and design principles, integrating performance benchmarks is equally vital. This means looking at how the code behaves under different conditions and ensuring that it meets predefined performance standards.

By weaving performance considerations into code reviews, developers not only identify potential bottlenecks or inefficiencies early but also foster a culture where performance is

a shared responsibility among the team. In practice, this involves analyzing how various components of the code interact, measuring execution times, and determining how scalable the design is for future needs.

Equally important is gathering feedback from the users of the platform. This real-world insight is invaluable as it provides a direct line to how the application performs in everyday scenarios. Engaging with users helps capture their sentiment, unveiling their experiences, both good and bad with the IT platform being created. It is an opportunity to learn about actual issues like slow response times or system crashes, which might not surface during internal testing. This feedback acts as a guide for developers to hone their applications, ensuring that it aligns with user expectations and needs.

Incorporating performance into individual roles

Performance should not be the sole responsibility of a dedicated team or an individual. Instead, every team member, regardless of their specific role, should consider performance from the outset. From UX designers considering load-times during design, to developers considering efficient code, and testers ensuring the software's performance under various situations, everyone has a role to play.

Encourage ownership and accountability

Encouraging developers to take ownership of the performance of their code and applications they build fosters a sense of responsibility. This will inherently push them to produce high-performing solutions, as they know that they would be personally accountable for any performance-related issues. Fostering a performance culture involves combining awareness, education, clear goals, routine reviews, personal responsibility, and team-wide participation. With these elements, you can build a development team that values and prioritizes performance, helping you to deliver superior software solutions.

Consider performance from the very beginning

From the inception of any software development project, measuring performance carries notable advantages that can lead to immense long-term benefits. This strategic move plays a pivotal role in proactively identifying potential performance issues and mitigating them well in advance. Having this performance-centric mindset from the get-go fosters an environment where the aim is not just to complete the project, but to shoot for high quality output every step of the way. Performance measurements provide rich insights that can be used to identify sluggish components or areas in the process that need optimization. This not only reduces the risk of performance bottlenecks during full scale operations, but also minimizes unnecessary expenditure and effort in rectifying these issues at a later stage. By focusing on performance from the start, you are more likely to deliver an end-product that offers a superior user experience, thereby leading to increased customer satisfaction. Faster response times, less errors, and an overall smoother experience are key indicative

Measuring and optimizing performance from the start can result in substantial time and resource savings. By avoiding late-stage performance tuning and last-minute firefighting, you will be able to focus on delivering innovative features and enhancing functionality, thereby maximizing productive output. Having performance at the forefront from the start of a project cultivates a culture within the development team where performance is prioritized. This not only prepares the team for robust performance engineering in the present, but also keeps them ready for efficient performance handling of future projects. It should, therefore, be a best practice to measure performance from the starting line of any software development project. It instills a valuable habit that encourages the pursuit of quality, helps save valuable resources, and builds a performance-first culture within the team. Once you achieve organizational change to put performance front and center, product quality will help drive and keep revenue.

Establish clear performance requirements

Understanding your performance requirements is vital for the success of any software development project. These requirements provide a clear measure of what the software product or application should be able to achieve in terms of its performance. It is a central part of performance engineering and carries immense importance in ensuring a high-quality end product. Without clear performance requirements, your development team may not fully understand the performance-related needs of the end users. This could result in the development of software that fails to meet user expectations or the needs of the business, leading to increased costs, delays, and potentially, loss of customers or business opportunities.

Creating quality gates for performance is essential in any software development process. This measures the performance of software against pre-set criteria and prevents any code from progressing through the software delivery pipeline unless it meets stringent performance requirements. In doing that, it will ensure that the software product can withstand real-world conditions and meet the performance and usability expectations of the end-user, creating an optimal user experience. Quality gates provide meaningful data that allows for evidence-based decision making in the development lifecycle. By focusing on actual performance metrics rather than assumptions, teams can make precise, informed trade-offs between performance, functionality, and time-to-market. The first step to setting up quality gates is to define performance thresholds.

What constitutes acceptable or unacceptable performance should be established clearly. Relativity can be detrimental in this sense, so consider response times, throughput, and resource utilization while creating these standards. In order to ensure that your software meets performance requirements, decision-making processes should be evidence-based. This means utilizing data acquired from performance testing to uncover areas of weakness or potential for improvement in your software. Once areas for improvement are identified, tangible actions can be taken to improve them, reinforcing the effectiveness of your quality gates. After establishing detailed and quantifiable performance thresholds, the next

step is to implement them within your development process. This can be done through various methods, such as automatic alerts for when thresholds are exceeded or integration within your testing suite to flag potential performance issues early on. Upon crossing the thresholds, the software would either need to be optimized or redesigned.

The project iron triangle

When we talk about the process of building fast IT platforms for great performance at a high velocity, one of the most important concepts to understand is the project iron triangle. This is a concept adapted from the political world and moved into IT for describing project relationships to describe the yin and yang of managing products. Its roots are based in politics in the *United States* to describe mutually beneficial relationships between a government agency, an interest group and a congressional committee. The triangle in an IT context describes the attributes of an IT project. It is an important tool that has been around for a long time.

Over the years, based on experiences on different projects, we can say with confidence that it is the most important concept to understand when learning to manage delivery of any IT platform. It describes the interconnected relationships that are mutually dependent on one another for building a product and how one impacts the other. Most organizations, at one point or another, have tried hard to break the relationships in the triangle. To date, they have been unsuccessful. When you see organizations rushing to push something out the door and it has a high defect threshold with low quality, we know that this is another example of trying to break the unbreakable project iron triangle. If you see an organization, try to have golden unicorns, people that wear all of the hats vs. experts in different roles wear competence and quality is higher, this is another example of trying to break the iron triangle.

One can consider the relationships expressed in the triangle as laws that cannot be broken. The triangle consists of three attributes which, when showcased together, represent the most important aspects around managing product delivery. On one side of the triangle is cost, another is time, and the final leg of the triangle is scope. Look closely at what is in the center of the diagram showcased in *Figure 2.1*. Good performance is only achieved consistently when the proper processes at gating is in place, which only occurs when these other items are equally balanced. An organization's utopia would be to be able to build an IT product that performs very well, one that is perceived as high quality by its users.

When discussing performance, remember that perception is more important than anything else. The optics around perceived quality is what is important to revenue. Most organizations want to create platforms that can easily adapt to changing requirements quickly for a business trying to obtain a competitive advantage in the industry. Hence, all organizations in this situation want to create an IT platform that has great performance at a low cost very quickly while still satisfying a creeping scope and changing requirements from the businesses that are watching competitors so closely. This, of course, is not practical

or costs around capital expenditures and resource costs. If you reduce time, you need to reduce scope, which increases costs.

As an example of an attempt to build an IT platform fast and cheap, let us say that you are building a platform and that you want to find the cheapest resources available in the world to build the platform for you. You find some resources on a small island in a different time zone that will work at a very low hourly cost that keeps labor costs low on the project. So, now you should be able to build the platform cheaply, right? But what you find is that with these new resources is that you have communication challenges, product learning challenges, challenges understanding business processes, inherent unforeseen cultural biases, governmental rules, regulations, sponsorship visas, or other requirements in addition to people dependencies across locations. You encounter timing challenges working across time zones, and challenges around skill levels, and motivation problems because they are not employees wanting a career. Building synergy across time zones where people are energized and in sync with one another also becomes challenging. You find that you now need much more time to achieve a good quality product. So, where you took an approach to try to save costs with that being your goal, you inherently caused much lower quality and inefficiencies and task completion delays. You impacted time which is the other side of the triangle. You realize that maybe you can hire more of these resources to get things down faster but you realize, now you need more communication, more management, more process, and thus more cost. So, you have not been able to go both fast and cheap like you had expected.

Let us say you have scope that keeps on increasing from the business incrementally, you leverage experienced employee resources that know the product, business needs and the technology, but the target keeps moving. You express your desire to be adaptable to change which makes your stakeholders happy because they can keep up with a competitor, but what you find is that, regardless of the processes followed, when scope increases so does cost and time. Thus, now that you realize that time is going to take lower using more junior resources, as a project manager, you try to drive people harder, work them more hours, but you find when that happens, quality and performance go down much further, more steps are skipped around quality, performance testing and quality gating is not done or good practices followed. More mistakes are made resulting in a large amount of technical debt and unhappy customers. The delivery, in this case is a slow platform because performance becomes secondary over features in many instances, though one could argue that in a well-designed platform that performance is a feature. Ultimately turnover occurs within the organization. When this happens, you have to hire and bring other resources up to speed, which takes two to three months on average. This is where the model fails. Where cost was thought to be saved, it impacted both good performance, functional quality and time for the product.

As shown in the underlying *Figure 2.1*, achieving good quality and performance requires balance across the iron triangle. You cannot skip on what is required to create a high-performance product by going fast and cheap. But is there anything we can do to still go

velocity and great performance? Is it just not possible to do? In the past, we have seen many project managers try to predict timelines based on resource numbers. For example, if an organization can create a product in nine months with one individual, then it must be possible to hire nine people and complete the effort in a month. Perhaps, we can throw more bodies at the issue. Experience over time tells us that this is not a viable strategy because all products have dependencies and not everything can operate in parallel throughout a project. With more resources, it means more coordination and more communication and synergy that must be established. The more resources that are involved, the more overhead around managing the resources is required. But that being said, there are a few ways to accelerate product delivery. One, is that you can create products quickly by throwing people at it and incurring the communication, coordination and overhead challenges. So, you do take a hit around quality and the overhead around communication which impacts the scale, but you eventually meet your goal of delivery. You break out tasks and parallelize as much as possible and then try to identify synchronous dependencies. Refer to *Figure 2.1*:

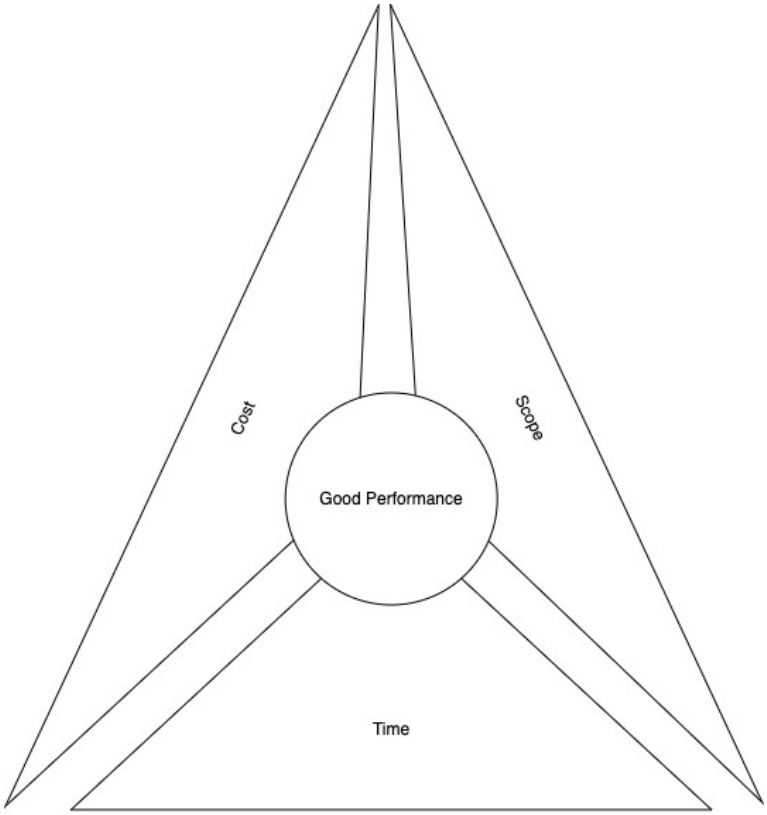


***Figure 2.1**: Project iron triangle*

A lot of work goes into this and the quality is generally pretty low but you can quickly ramp up teams quickly, because you are not looking for the rock stars around product delivery, you are looking for more junior resources and a couple of more experienced folks

to help guide them. This is one approach that is commonly used today in product delivery in large organizations, with many efforts going on. Instead of efficiency, synergy, and low head counts, process and higher head counts prevail and a large amount of technical debt and code that is costly to maintain. Overall, this process, results in higher costs but delivery does happen.

Software development methodologies

Let us talk about the different most prevalent software development methodologies in the industry:

IT waterfall software development lifecycle

There are many ways to deliver IT platforms. Software over the years have had processes that are more waterfall in nature where there were well thought out steps along the way. When doing a waterfall process all of the requirements are created upfront and reviewed. Communication is good for everyone working not the project and signoff occurs at each stage and communication is clear throughout the process. Documents around everything for the product are kept up to date and parties all signify on the delivery of different artifacts. The scope is fixed, the cost is fixed if the project remains on schedule and delivers as expected. As long as scope does not creep, which was the largest challenge, the waterfall model delivers some very robust platforms. The problem with the waterfall was that the scope could not change. Businesses could not change requirements on a dime or adapt scope throughout the effort. For large projects, it would sometimes be possible to get to the end to only then find out that the requirements were not properly understood. Many times, business needs would change during at 12-month project and 6 months of cost was poured down the drain that could not be recovered. That being said, waterfall is still used heavily today and is a great approach for small teams delivering products using mature known technologies. Here are the phases of a waterfall process:

- Requirements
- Design
- Coding
- Testing
- Deployment
- Maintenance

In the traditional waterfall methodology, development processes flow sequentially from one phase to the next, akin to cascading water over a waterfall. Here, it is crucial to make performance a consideration from the very beginning. In the context of the waterfall development model, performance engineering carries critical importance. Because every phase is distinctly separate, performance considerations must be taken into account

from the beginning. In the requirements phase, performance targets should be clearly established. As the design phase advances, architects must ensure that these targets can be achieved with the proposed system. Performance testing and tuning logically fall in the implementation and verification phases. Performance testing in this model is typically thorough and extensive, accounting for various user loads and data volumes. If performance targets are not met during these tests, the project steps upstream to the design or even the requirements phase to determine the problem source and address it. Waterfall development essentially follows a cascading approach where each phase of a project must be completed before the next one can begin. There is no going backward in this model ore adapting to change frequently. If a problem or a change request arises during the implementation, the project must cycle back to the design or requirements phase. One challenge with waterfall and performance engineering is the late in the life cycle realization of issues. Since all phases are sequential, problems are sometimes identified far into the project, necessitating changes to earlier stages. Consequently, performance improvement modifications can be expensive and time-consuming because the rigid structure means stepping back many phases. Yet, the waterfall model also presents unique opportunities. It allows teams to anticipate and solve issues before they become more significant problems. With thoroughly outlined requirements and robust design architecture, potential issues can be proactively addressed and resolved in the design phase, often before substantial coding has even begun. This brings us back to the principle of stepping left as early as possible in the process. Starting early, waterfall teams have the opportunity to make significant performance-improving decisions in the early stages.

Finally, because waterfall organizes production into discrete phases, it is easier for teams to focus on performance during each individual phase. Each phase becomes a period of intense focus on its particular aspect of the software's operation, including its performance.

For waterfall, the emphasis on performance engineering ensures that the software's performance is, to a large extent, predictable. It becomes easier to maintain or even improve performance because of the comprehensive knowledge that allows in-depth troubleshooting. Performance measurement should start as early as the development stage. This proactive stepping left approach allows you to identify and address potential issues before they become problematic. Ultimately, stepping left, sometimes referred to as shifting left, in the development lifecycle to build in quality at lower costs means adopting a forward-thinking mindset. It is about transforming an organizations quality approach to architecture, design, data and performance, making it an integral part of our day-to-day operations rather than an afterthought. This shift not only enhances your performance metrics, but also fosters a culture of continuous improvement and innovation from the beginning of the product lifecycle. Let us talk about the steps to get started incorporating a stepping/shift left strategy into an IT platforms development.

Steps to get started and assure that the product is developed with a quality mindset:

- Begin by first establishing the non-functional requirements around performance for the platform along with expected resiliency requirements. Architect, design,

and code to these defined non-functional requirements, execute proof of concepts around testing performance if needed. To meet requirements, define coding standards and best practices to be followed and audited in code reviews around good performance coding patterns. Establish expectations for how fast methods should execute, if they should be multi-threaded, use parallelization, async programming or other patterns when beneficial. Specify design pattern requirements around data access layers and databases such as relational or NoSQL, using stored procedures, coarse grained requests across the network, TTL caching strategies, etc. Establish good web practices around web pages, low page weights, low number of requests across the network, minification, compression, combining small files, image optimization, deferring/async JavaScript, etc. Minimize object to relational mapping and data transformation layers as much as possible for the IT platform.

- Incorporate performance testing and optimization during the implementation stage. This means profiling and performance test automation around the codebase as it evolves. This is critical to be successful. As software is profiled and bottlenecks are defined, optimize both the software and runtime environments, JVMs, Kubernetes, on-premise environments etc. to assure optimal performance and to be able to meet the requirements established. The hardware should always be optimized to the specific software running based on its thread patterns, memory and CPU profile. When new software is released with significant changes, the runtime environment configuration will continue to change over time.
- Continue to regularly monitor and enhance performance in production using both real-time user metrics and synthetics to identify optimization opportunities. First, regularly verify production against an environment where capacity tests are executed, an environment that should match production 1:1. If it does not align environments or mathematically extrapolate differences in data, infrastructure, configuration, etc. Identify where the platform has slow performance in production and replicate the issues in lower environments and fix them.

Establishing good performance requirements upfront is utterly essential in the waterfall methodology due to its linear nature. Once a stage is completed, it is considerably difficult to go back and make changes. Remember, in the waterfall model, changes later in the development cycle can be costly and time-consuming, which makes early performance considerations all the more crucial. When determining performance requirements, you can benchmark against a competitor's platform to understand the current standards and expectations. If a suitable competitor platform is not accessible, leverage resources like **https://httparchive.org/**. HTTP Archive provides a trove of data about web performance across the top sites on the web.

Here is how you use it:

1. Navigate to **https://httparchive.org/** and review the REPORTS section.

2. Find the **STATE OF THE WEB** report
3. Choose a Lens of the Top 1,000 websites and a date range and click **Update** to run the report.

Make your requirements better than the values on this report. Leveraging this information early on will allow you to set ambitious, yet attainable, performance metrics that can guide the development process in a performance-oriented manner. When it comes to performance engineering practice, it is crucial to understand that it cannot be an afterthought.

IT iterative software development lifecycle

Consuming an iterative software development process at its core allows for fundamental consideration of performance at every phase. Let us dig a little deeper to understand why this is so crucial.

Method	Advantages
Waterfall	All phases clearly defined, however performance measurement only possible post-development which may lead to significant rework.
Iterative	Preemptive performance measurement, constant enhancements, integrated feedback loops, early identification of issues.
Agile	Flexible and adaptable but the focus on delivering functionality may sometimes overlook performance metrics.

***Table 2.1**: Software development lifecycle types*

As can be seen in *Table 2.1*, the iterative approach discussed next empowers you to set ambitious performance targets while giving you the flexibility to make continuous improvements and measure performance in real-time. It also lends itself to continually to not only innovate a product with a fixed set of technology, but to continually innovate the technology itself driving great efficiencies and reduction in software costs many times around discoveries made. The iterative software development process handsomely pays off in the long run by saving rework efforts and delivering a superior, performance-driven product. The iterative processes come in all shapes and sizes. There are too many to mention. Iterative delivery processes came into the picture for software delivery after organizations needed to adapt to change more frequently without being impacted financial for work stopped in-flight like the waterfall process. The nature of iterative software development processes provides a unique opportunity to measure and improve performance from the ground up. Unlike the traditional waterfall model or even the Agile model, iterative development is akin to a loop; with each iteration, you are able to gather data, make enhancements, and measure performance before moving to the next cycle. Always remember that perfect performance should always be considered a journey, not a

destination. Every iteration presents an opportunity to better understand the software and improve its performance. Since every iteration is an opportunity to measure performance, it becomes a proactive practice to include performance metrics early on in the software development process. This means that rather than waiting for the product to be entirely developed before analyzing how it performs, you actively measure it at every stage. You can do this by setting targets for individual features and modules and measuring their performance through specific indicators. Iterative software development also offers the advantage of prototyping with different latencies. Usability studies can be conducted to record user responses even before the product is fully developed. This early bird approach allows you to understand the impact of performance on the potential revenue of the final product.

In this model, when comparing to the waterfall process, the waterfall steps were broken up into small subsets of work. Stakeholders would receive frequent feedback. Prototyping occurred often in the project to flush out requirements between IT and the business using high level tools that could rapidly create products. Though we can remember when this process was popular industry wide that the business leaders would many times think that the product was already done when showing them a working prototype. The one disadvantage of using high level tools like PHP or slower interpreted computer languages like Python with high levels of code abstraction for rapid delivery, is that the business may think the work is already done. This methodology was a very good balance between lower overhead around process, still having important milestones and helped keep cost at bay. In this model, the granularity and frequency determined the ability to adapt to scope changes, but it is a very successful model. This is the model to use when technological innovation is involved that is being learned by the development team. It is good for large projects with high-risk features where the requirements for the project may change later on in the development cycle. From my experience over the years, iterative approaches to software delivery are the best and are good for innovation and continual improvement. For many years, people coined the term iterative waterfall but regardless of the specific iterative process chosen, this is a great process for product delivery. It moves away from large sets of documentation and instead into prototyping, presenting sketches, and blueprints to the users and receiving live feedback many times before any product development even begins to take place.

IT Agile software development lifecycle

Finally, after a successful iterative model like the **Rational Unified Process** (**RUP**) which was a popular iterative model, then came Agile. Performance engineering is of paramount importance in Agile development. It entails the measurement, evaluation, and optimization of performance throughout the software development process. In Agile development, frequent iterations make it feasible to evaluate and enhance performance at regular intervals. It is important to foster a culture of performance from project inception, thus mitigating any performance risks that may arise. Performance engineering is an integral part of Agile development methodologies.

approaches, Agile is an iterative process that favors frequent inspection and adaptation. Integrating performance engineering practices into this Agile approach can result in more efficient, effective, and reliable software. Agile development emphasizes continual testing to ensure changes do not degrade system performance. Each sprint provides an opportunity to test and optimize performance. Since Agile allows for constant code and feature improvements, it is an ideal framework for integrating continuous performance testing. During Agile backlog grooming sessions, user stories are often fleshed out and prioritized. It is important to include performance considerations as part of this process. Prioritizing performance-related stories and making performance acceptance criteria explicit will ensure the team keeps performance in focus. Performance engineering metrics can be integrated into Agile sprint planning. When the team is deciding what work to commit to in the next sprint, predictions about performance impacts can inform these decisions.

Agile Manifesto

If you have worked in the world of software development for a while, you have likely heard the term Agile Manifesto. It is an important guide that became a blueprint regarding how many teams approach projects. The manifesto is a statement of values and principles designed to facilitate better ways of developing software through collaborative methods and able to adapt to change requirements quickly. Agile processes have good points and bad points but very much success depends on balanced interpretation. For example, one of the principles is to have working software over comprehensive documentation, but that does not mean no documentation which is where many teams faulter. The Agile manifesto introduces four core values and twelve guiding principles that shape Agile methodologies.

Here are the core values:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

Below are the twelve guiding principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity the art of maximizing the amount of work not done is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjustsits behavior accordingly.

The Agile Manifesto, when it came out to help drive Agile in the workplace, emphasized the concept of people over processes, focusing on collaboration and elimination around a lot of documentation. Think about the time in history when this was first introduced and how good this would sound. People were writing large documents that were hundreds of pages long in many cases to make sure stakeholders and developers, testers, IT leaders were all on the same page. Understanding what was going to be created holistically did drive quality into the product which is one of the disadvantages of Agile. It was this understanding that ultimately led to much less rework, less defects and technical debt overall.

But even with the potential downsides at the time, when this first came out, people in IT were so relieved that they no longer had to write lengthy documents to fixed templates which reflected the software design or testing strategy. We can all remember a time when people would say that the source code was the documentation and nothing else is required as Agile emerged. People originally took the concept of no documentation to heart right away because it was such a relief from the process they were used to following.

Requirements were no longer phone books of information to put everyone on the same page. The downsides of course were that a lot of interpretation around the new process was going on and no one really knew the right way to balance the concept of no documentation. To this day, there is still a limited understanding of the product being created with the Agile process and the limited information put on wiki pages is many times not kept up to date with any signoff or acknowledgement that it was read or agreed to. Agile requires exceptionally good communication to be successful and extroverts who communicate well, which for large projects is not something that really can be guaranteed when it comes to hiring talent. The best technical talent is generally the analytically thinking people who are not as collaborative and outgoing as generalist. This process has lent itself to delivering

low quality software and generating a lot of rework. Though the initial intent of the Agile methodology was to eliminate a lot of documentation, at least 25% of the time working on an Agile project is managing tickets and tasks for requirements, rework, defects, and other delivery tasks, many times much more fragmented documentation in a ticketing system than what would be seen in a waterfall or iterative delivery model. One of the problems with Agile is the people concepts in the manifesto, if you are in a political climate like most large organizations have, then people tend to not share information as well as a fixed communication structure where everyone receives the needed information to their jobs well.

This inherently breaks the Agile process because it relies on great communication and collaboration to really be successful and giving individuals the ability to control or manipulate the information and message, ultimately defeats the success. Also, with Agile where tasks are at such a granular level in small sprints of work, micro-management is required in many cases and resources far left in the process do not take ownership for what is being pushed right into the process. People become robots in the process driving towards a task delivery and forget to think for themselves around better ways to do things or innovate. The reason for low quality in Agile processes is because if something does not work, it just means that a ticket would be generated and put into a backlog which would likely go to someone else to address at a later date versus the individual that generated the defect. So, creating bad code is unfortunately encouraged though that was not done purposefully when Agile was developed. Many times, to make a delivery date, the person farther left in the processes does not do the due diligence and throws the software right in the process quickly. They may have generated something that never worked. As a result, more defect tickets are created that have to be managed and rework levels go through the roof. Tickets also tend to be created as defects that are new requirements or to stall the delivery process and buy more time. To try to still deliver in these situations, project managers or owners will try to avoid the work, keep things broken and say it is just technical debt. This inherently creates platforms that work with very low quality. In Agile processes, we see defects tracked and who caught the defect but people rarely pay attention to who is consistently creating the defects which is an important metric to track to push quality left in the process. When quality if fixed far left in the process, the costs are low and everyone wins.

One advantage of the Agile process is that it does a good job at adapting to continual change and keeping the business and IT aligned more often which has been the largest reason for adoption. Agile built upon iterative processes and shares some advantages around being able to adapt to change at an even greater level. Agile breaks down a project into fine granular pieces even further than processes like RUP and generally management software is required like a ticketing system such as Jira or Rally to break down units of work by an individual requirement via the concept of Epics and Stories which are how requirements are delivered over a document with a standard template. The problem with Agile is that many times you will find that unlike RUP and iterative processes at a courser level, the big picture gets completely lost and granular tickets become difficult to manage

and prioritize. So, management can be more time intensive, communication can be more difficult as well because the unit of work is so small. It means frequent micro-management around small time windows and tasks. Anything not gotten to ends up flowing over into another sprint in the delivery cycle to becomes technical debt for the effort. Principles around Agile mean it absolutely requires good communication and collaboration to be successful which unfortunately is usually what breaks the process. It means building projects around motivated individuals and continuously delivering even when project requirements come late into the picture which does not effectively use all resources in the organization to the best of their ability. Inherently, bias comes into play as well around the concept of people over processes and this generates more turnover and angst because people want to be on a level playing field around communication.

Agile was meant to an improvement to project delivery and it gives IT product managers the ability to never complain about a changing scope. But since Agile processes came into the picture, quality of software platforms has been reduced significantly in the industry. It gives organizations the ability to adapt to changes in scope at any time which is great for any business leader, but the delivered solution in my experience is low quality. To this day, the highest quality software is still built with waterfall and more innovative products that require good quality developed with iterative solutions with experienced individuals and small teams embracing good synergy and most importantly, much more innovation. Iterative process solutions are the best-balanced processes eliminating the disadvantages of waterfall around scope creep and the disadvantages around required communication and lack of ownership seen with Agile.

IT DevOps

In the realm of software development, DevOps is a driving force, reshaping the industry with its mindset of blending DevOps into a seamless singular process. As DevOps takes the world of software development by storm, an unerring knowledge about the role performance plays within this paradigm shift becomes of absolute importance.

DevOps is based on the principle of frequent yet reliable software releases, which ultimately enhances the system's performance. Performance here is not just about the speed and responsiveness of software operation but its overall availability, reliability, and smooth user experience. The primary goal of integrating performance into DevOps culture is to ensure that the operations team does not have to scramble to fix performance issues after the software is near completion or after a version is already out in the wild. When performance becomes a central focus from the inception of the software development process rather than an afterthought, it paves the way for software that runs efficiently with fewer speed bumps to contend with. In DevOps, performance is not just an attribute of the system, it is a feature. It is something that should be defined, designed, implemented, and tested just like any other feature. Performance concerns should be addressed right from the planning and design stage, through coding, and into testing and deployment. It brings an added layer of resilience to the system and improves overall user experience.

DevOps practices such as **continuous integration** (**CI**) and **continuous deployment** (**CD**) attain new dimensions with performance as a focal element. With continuous performance optimization, performance improvements and adjustments are not just one-off tasks. Instead, it becomes a recurring habit during the life cycle, ensuring that the software continuously improves and evolves. It creates a snowball effect that incrementally boosts the performance of the software over time.

Project delivery resources

Over the years, we have seen organizations hire fifty people with a lot of process, communication, oversight, tickets, granular task and project management, leadership etcetera where resources are junior, mid-level folks that are lower cost. Some organizations have the additional burden of working with a great deal of diversity and communication challenges across cultures leading to rework. We have also seen many organizations hire four or five senior experienced people that did multi-tasking across the same level of work as the fifty junior resources, with much higher quality and less overhead. Both approaches will deliver products. The difference is that with the fifty more junior resources, heavy process is required where the overhead with the five senior resources is much less without much process at all. That being said, unlike what you may think, sometimes the fifty are easier to find than the five because those five senior resources have a lot of options with a lot of different organizations trying to obtain their services. The adoption of virtualization and being able to obtain the best IT candidates from all over the world these days really prevents this issue. Time has shown us that the most productivity and the least turnover exists when employees can be working during commute times extending their day or balancing their work and personal life. Flexibility makes an employee want to give their best. When taking this delivery approach of using senior talent, quality is higher, innovation is nimbler and customers are happier because they receive a high performing product that is fast. The product is also much cheaper to maintain with less code, more consistency in design and coding patterns.

Hence one approach is, instead of hiring twenty or thirty junior resources, an organization hires four or five very experienced senior people that can create automation and efficiencies that consistently build quality into the product with little process and little technical debt. Taking this approach eliminates the process overhead, communication overhead, management overhead and the quality of the product is many times higher than throwing more headcount at the project and the challenges around that approach to product delivery. You pay more for the experience and efficiency, but with much less head count. You put these four people next to one another and attempt to create a lot of synergy versus a lot of process. You start by asking them to create a great platform, or platforms, that are more innovative than your competitors. The four people are paid and treated very well and are given latitude around creativity and flexibility with the design process. As a pre-step to building the product, the requirements are laid out very loosely initially allowing for the four individuals to create a prototype reflecting on what they would consider the most innovative product using the latest mainstream technology just outside of the hype

cycle. The hype cycle is an IT industry curve that reflects the adoption of a technology in the industry. For example, has the technology been accepted by the industry or is it just being marketed as something that will solve all problems with no real-world adoption. In other words, has it been vetted showcasing real-world usage and proven itself. It is best that these four folks are employees and not contractors because they will be much more committed to creating a great product and learn the business.

Technological innovation

Beyond product innovation often seen in hackathons or compartmentalization around innovation, there is process and technology innovation which is many times is overlooked in the product development process. We think in terms of product ideation only. When technology and process innovation are considered, what happens is efficiency and a side effect is great quality with a platform that is easy to maintain at a low runtime cost. As part of the solution one of the key requirements looked at is how to create a solution that is efficient and quick with great quality and performance. For example, let us say the standard for web development at an organization is to use Angular JS which is a popular web framework used in IT platform development. But through some prototyping, through research you discover that two other web frameworks named React and Vue have a concept of a virtual DOM allowing for much more in-memory fast DOM interaction for a website, improving web performance tenfold. Elements on the web page can be interacted with through programming at a much faster rate than ever before. This may mean that you can load web pages in one second versus four seconds. You do research and find that most of the industry has already gone this direction.

Perhaps you may also find through your prototype that you were able to use a solution and code the capability in less than half the time using one solution over another and the process to build and optimize the application was easier than the old process. So, innovation is not just about product innovation and ideas for products, it is about continual technological innovation and streamlining efficiencies in the product creation process itself through technology and process changes. On a side note, regarding process, one of the most important considerations when choosing application frameworks is not deciding on what product has the most features, but instead which product has the features that you will actually use with performance and benchmarking being the top consideration. We have seen frameworks being used today by organizations where eighty percent of the framework is never leveraged but yet that bulky unused code is being sent down to browsers or being loaded by class loaders in the JVM or other runtime environments taking up memory and hurting performance significantly. If you consider performance as a key requirement in framework selection, you will find that those products are optimized with just the code needed, kept lightweight and efficient without the bulkiness seen with other frameworks. Another consideration is around the debug code embedded into these large framework products when deploying to production. Debug code is great int he lower

environments but it adds twenty percent to the target operating runtime code. This slows down the code loading, uses more memory and ultimately impacts scale and performance.

Keeping the process small initially

There is one other process that we see organizations follow. Some organizations come up with the concept of research labs or incubation labs around innovation or product initial creation. An organization will hire the four senior people in the scenario above and have them build the product using innovative thinking and innovative technologies. Once it is delivered and customers are on board, it is handed it off to junior and mid-level lower cost resources to maintain going forward. Hence, go with mature experience and synergy with only a few folks, keep it small with great performance and quality with flexibility around technology and innovation. Then once the product is delivered, hand it off to a support team of lower cost junior and mid-level resources considered the team that will handle modification and support going forward. Important to have the junior folks create the documentation and follow a well-documented process to support and maintain it and work with the experience resources to fill in any misunderstandings. Have a good project transition and documentation strategy for when this occurs.

Some organizations will refer to as the project creators as the build team and the people supporting the application as the run team. But regardless of terminology, once the product moves to the support team, rigor needs to go into the process since experience and synergy is no longer being relied on. It is in this process that performance needs to be considered at every step for quality gating the product through its ongoing lifecycle around changes or modification. It is also important to monitor the product and have the necessary alerting in place. Change and delivery processes must be mature when the product gets to this stage.

Staying ahead of the competition

As far as fast product delivery and staying ahead of competitors through innovation, let me just say that this is the best process to follow and will deliver a high-performance IT product that performs well. You build a high-quality product with a few resources in a vacuum without the noise around processes, restrictions, etc. in an organization using highly technical people with synergy given them a lot of flexibility. Then you transition the product once fully created to the team with process to manage it ongoing after delivery in the organization. You get the benefits of both models, high cost few resources, synergy, innovation, etc. and then lower cost resources to maintain it ongoing. You have overhead now to support it, but the product quality and technical debt at initial creation is almost nothing and the codebase is fifty times lower than if you would have delivered with the junior resources and high process approach initially which keeps costs down and quality up. For larger organizations with enterprise architecture groups, too much flexibility can be scary. It means more technologies to support and put rigor around.

As someone who worked in enterprise architecture for many years, let me just say that this groups most important function is aligning the business domain with the delivery or technology domains for solutioning and knowing the capabilities of each area is key to letting that happen. It is important to not follow an Ivory tower approach to enterprise architecture. Follow concepts around technology, letting the innovation and flexibility happen is key to staying competitive and that means innovation for people that are using technology day to day. Then drive best practices around technologies adopted through this innovation or incubation labs and tie those capabilities back to aligning with the business needs. We have seen so many enterprise architecture groups dismantled because they were thinking in terms of a tops down approach around constricting innovation and identifying standards versus being able to adapt to change and let the innovation happen. Full discloser, at one time I was one of those folks all about standards. But in today's world, instead of standards around limiting technology, it is better to focus on standards around design patterns and hiring practices, industry acceptance and licensing is the key to being competitive and safe.

The key to being successful when hiring a small innovative senior team to build a product is to understand that these folks are leaders in the space and trust them. This is an important, because what you would never want to do is let innovation happen with inexperienced talent anywhere and everywhere in the organization, that would be a recipe for failure. A well-designed platform with the least amount of code following good performance design patterns from a small innovative senior team will always be fast out of the gate. It will also make customers very happy because regardless of the feature set, perception around speed translates to perception around quality which is an interesting behavioral trait.

Total quality management

It was not that long ago that a concept around product quality called **total quality management** (**TQM**) was mainstream in the industry. This process meant continual improvement around performance, processes, technology for the benefit of the customer. If you were an IT shop or an auto repair shop, it did not matter, you had processes you followed to deliver a product or service. TQM is about continually doing things better and when it comes to performance engineering, TQM is an important concept. Some organizations implement this by flow charting their process. Let us say you implement a process that builds software with great performance practices, but you need to improve it. Perhaps, you feel that there are too many steps, maybe it is too complex or you feel that performance measurement is not occurring often enough in the process. You do this by flowcharting the existing process and finding ways to either remove boxes, making the process more efficient, or adding different flows to build better quality into the final product. But the point of TQM is that it is ongoing and change is ongoing. It is continual improvement. It is important to be continually looking at how things are working in an organization to drive efficiencies which translates to revenue. The continual improvement of the processes is something that continues to happen and allows platforms to build higher performing and higher quality systems.

IT performance cycle delivery process

The IT performance cycle in a standard delivery process which will assure that an IT platform has fast iteration after iteration. The architecture is created with considerations around flexibility and extendibility upfront. Some great architecture patterns have evolved over the past twenty years and they are proven. Using good patterns and taking into account performance upfront assures a great platform out of the gate. The database is designed with flexibility around relationships anticipating change and build with efficiency, the needed indexes, referential integrity and speed. Normalization versus denormalization is considered based on what the datastore will reflect and the parties using it. The code is created with minimalism in mind, efficiencies with small amounts of code using design patterns that allow for fast development and low maintenance runtime costs. Error handling is minimized and caught through automation. The runtime environment is optimized to the code itself. In other words, if a thread in the code takes up 10 megabytes of memory and we have 1000 megabytes of memory, we restrict the number of threads making requests to the JVM to not be able to overload the JVM. Then we precompile the code upfront or optimize the **just in time** (**JIT**) compiler for the JVM to give the best performance. Then we look at CPU compute being used per thread sometimes referred indirectly as CPU time and optimize the processing rate of the CPU to meet the performance requirements of the platform with the garbage collection methodology being used.

When building platforms with high quality with senior resources, much more time can be spent on creating maturity and efficiencies because so much less time is spent on coding or rework and backlog items for the product's initial creation. Less process and tickets and the management required to manage those tickets is required. As code transitions from build mode to run or support mode, all builds of the software should be fully automated going through quality gating of both nonfunctional requirements, including performance, and also functional requirements. Builds should fail during deployment with the quality gates are not met. Nothing should make it from one environment to another without quality gates passing at each layer. *Figure 2.2* depicts the IT performance cycle which talks about the different stages of performance measurement throughout the software development life cycle:

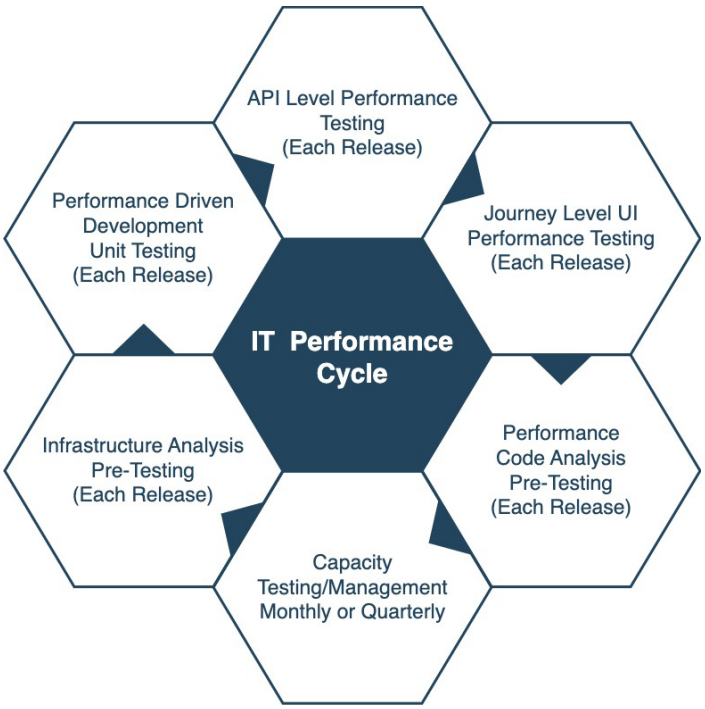

API Level Performance Testing (Each Release)
Performance Driven Development Unit Testing (Each Release)
Journey Level UI Performance Testing (Each Release)
IT Performance Cycle
Infrastructure Analysis Pre-Testing (Each Release)
Performance Code Analysis Pre-Testing (Each Release)
Capacity Testing/Management Monthly or Quarterly

***Figure 2.2**: IT performance cycle*

IT performance cycle

An essential component of modern software development is the IT performance cycle, which emphasizes **shifting left** or as we like to call it, **stepping left** since building a robust platform consists of multiple steps along the way. This means prioritizing early testing and quality assurance during the initial stages of product creation. By integrating performance tracking and testing from the very beginning, teams can identify and resolve issues early, ultimately enhancing the overall quality and efficiency of the development process.

However, the IT performance cycle does not stop at the beginning stages. It extends all the way to the right in the later phases of the development lifecycle ensuring that performance is consistently measured and optimized at every step. This continuous measurement enables teams to maintain high standards and swiftly adapt to changes, which is particularly crucial in Agile environments. The result is a robust, feedback driven process that aligns tightly with the principles and values outlined in the Agile Manifesto, leading to higher-quality software and more successful projects.

Performance driven design/development

In creation phases, consider performance patterns up front for software design. Prototypes

uncertainties regarding the design patterns being used. **Performance driven development** (**PDD**) practices where all unit tests include performance measurements using micro benchmarking frameworks like **Java Microbenchmark Harness** (**JMH**) and web analysis should be done for websites using tools like Lighthouse and Chrome Developer tools with performance insights. A good precedent is to assure that all methods in source code when ran execute within 100ms and all API clients execute within 500ms for **service level objectives** (**SLO**) at this layer far left in the process. Web pages should be interactive in under two seconds across all devices and browsers where the product will be used.

API performance level testing

Multithreaded tools like JMH combined with low overhead web service clients are used to do API load testing or leveraging tools like JMeter or Gatling or other solution. The best solution is one with the lowest overhead and least standard deviation for response time across performance runs. JMH because it is a micro benchmarking framework and was designed with low overhead in mind has a standard deviation that it contributes itself as only 2ms. It also has very low resource utilizations around CPU and memory whereas larger overhead solutions like JMeter or Gatling contribute to the standard deviation around 200ms on average and require much larger resource utilization because they are so both heavy from a class loading perspective and feature rich. But any solution will work as long as the response time contribution by the tool itself is understood and automation can be implemented using the tools against quality gates to check if SLO are being met. Neo-load is another nice load testing tool on the market that has built in SLA profiles. This provides the ability to quality gate against test paths used frequently in CI/CD processes.

Journey level testing each release

Tools like JMeter and LoadRunner or other tool can be used to load test the UI Journey as part of the release process. LoadRunner has the ability to test the frontend built into the tool using something called **TruClient**. JMeter does not have the ability to test the frontend itself so it can be awful in testing web applications. Frontend JavaScript or render times are not considered like the TruClient LoadRunner product. To overcome this limitation, Selenium or other tool must be paired with JMeter to measure frontend performance in combination with drivers for web browsers/devices or a third-party device framework like BrowserStack or LambdaTest. Tools like Selenium or even solutions like Playwright can use the performance timing APIs to gather performance data through the UI when the platform is under-load on the backend. Generally, the top ten UI journey flows are automatically tested for each release because these ultimately impact the capacity limits for an IT platform. But understanding how the latencies across devices/browsers is an important consideration around product latency.

Performance code analysis

Tools like PMD, Lighthouse, and performance insights in Chrome Developer tools or third-party websites like Gtmetrix can do analysis of the code base to determine if best practices are being followed. PMD has the ability to do performance analysis of Java source code by injecting rules related to performance to be leveraged when source code analysis takes place.

Here is PMDs documentation around performance rules being checked during analysis:

https://docs.pmd-code.org/latest/pmd_rules_java_performance.html

Capacity management/testing

Monthly or quarterly capacity testing is done to determine the point where SLO are being breached for APIs and user journeys. Capacity testing does not mean, let us generate load onto the platform until everything falls over, it is at the point where the application is usable or having an acceptable user experience. Once SLO are breached the capacity has been exceeded. Capacity management ensures that IT resources are optimized to meet current and future business requirements in an efficient and cost-effective manner. It balances supply against demand, which means not only meeting today's needs but also preparing for tomorrow's growth and considering the incoming pipeline of potential traffic and usage. By carefully analyzing and managing capacity, organizations can anticipate potential performance bottlenecks and optimize resource allocation. This proactive approach helps avoid downtime while ensuring optimal performance of applications and infrastructure.

Capacity testing, a crucial subset of capacity management, focuses specifically on determining the system's ability to handle anticipated workloads. It involves conducting tests to assess the maximum user load that your system can handle before performance no longer meets service level objectives for the platform. By simulating different load scenarios, capacity testing helps identify the exact point at which your resources become strained and resistance to load.

Effective capacity testing provides valuable insights into areas such as:

- **Scalability**: Understanding how well your system scales as demand increases.
- **Resource utilization**: Observing how efficiently your resources are used under maximum load conditions.
- **Thresholds**: Identifying critical thresholds that risk a slowdown or failure.

Engage business analysts and application engineers early in the capacity testing process. This fosters a collaborative environment where potential challenges are proactively addressed rather than reactive patchwork in production. Moreover, by integrating capacity management strategies with regular performance testing cycles throughout the **SDLC**, organizations can enhance end-to-end

capacity management and testing are not just preventative measures but strategic tools that contribute to maintaining robust and efficient IT service delivery.

Infrastructure analysis

A good infrastructure design and configuration can really improve performance for an IT platform. Infrastructure is analyzed in the lower and environment environments user load and continually optimized to achieve the best performance. This includes **content delivery networks** (**CDN**), reverse proxies, balancers, Kubernetes clusters, JVM health, API gateways, and even managed services where optimization is possible etc. Infrastructure is configured to support the load needed before any testing takes place. This significantly minimizes testing iterations. If we know that the platform needs to support 200 requests per second, we should have thread limits set to handle that load level. If we are balancing across two regions to achieve that 200 requests and we need to be able to failure to a single site and handle 400 requests, then threads, database pools, etcetera should be set to handle 400 requests. If we need the application to be cached at the browser for a web application or we need data to be compressed going across the wire, reverse proxies should be configured to do this. If we know that each thread takes up 20 megabytes of memory, then we should have enough memory configured in the platform to handle the needed load. Once a platform is optimized correctly, it should not be possible to overload it, it should instead, reject, redirect or queue traffic.

As shown in *Figure 2.2*, there is an IT performance cycle that reflects what is done interactively on a platform around performance after a platform transition into a maintenance mode or is in a support model where more features are being implemented in an Agile environment. When in the requirements phase, all functional requirements should specify their non-functional requirement around response time at a given bandwidth for a service and resiliency needs. For web pages, a browser type and version should be specified along with bandwidth and response times for page loads and interactions.

Design considerations and product scoring

In the Architecture phases of product delivery, the platform should be architected to balance flexibility and performance. When the code is in development, the coding practice should incorporate PDD practices. PDD is involves designing the software for a product taking into account performance early on. The goal is to get performance metrics down lower related to response times and potential error rates. This includes taking into account resiliency and designing in the necessary redundancies and tuning the solution to support failover at the failover points with minimal impacts to the responsiveness of the IT platform. It involves creating any needed prototypes to veto solutions. For every code release, including frontend code for web applications, the code should be analyzed for performance best practices and good performance design patterns. This is done through code walkthroughs and with code analysis tools like PMD and Lighthouse. It is good to score the process and base a threshold for released based on that score and not release

the software until a score level is met considered as low risk. In fact, I would say listing out criteria around maturity and having scorecards for teams as far as how well they are meeting the criteria is a great thing to do.

Let us give a couple of examples of this so that you can see some of the dynamics around designing fast application and the considerations taking place as part of PDD. A good relational database model reflects the real-world relationship around the entities in it. It is typically highly normalized. When a relational database management system uses a normalized database, it assures referential integrity and eliminates redundancy of data. In other words, using this type of data model stores the least amount of data in the most efficient way as far as resources used in the database. Almost every database administrator on the planet will want to use a highly normalized model because from a database perspective, it uses the least number of resources. But when it comes to designing applications for great performance, we have to make tradeoffs around database resource utilization and performance. The disadvantage of a highly normalized data structure is that many times complexity of SQL causes issues around performance. It can be easy to not join on the right keys and what occurs is full table scans of the data as part of the SQL itself. Normalized databases are a good idea when multiple different applications are hitting a data source and it will be a system of record for multiple operational platforms or it is a data warehouse type of database which is a system of record used to feed operational systems downstream. In this situation, each application must then optimize the model for their own needs using techniques such as materialized views or creating denormalized versions of their own data structures from the system of record. This is seen in the concept of data warehousing quite a bit where large highly normalized datasets are the system of record and then feeds from that warehouse are fed down into more denormalized operational platforms tuned for performance for a specific application.

When a data warehouse or system of record data storage is the same database as the operational model, this can cause quite a bit of challenges for performance. Applications to achieve great performance must align how they are requesting the data with the data model itself. In this situation, regular refreshes of materialized views where SQL is running that is precomputed and physically stored ahead of time and updated on a recurring frequency and then the SQL to access them from an access perspective has been simplified. Materialized views allow for simplification around the model and efficiencies around performance. But let us say the model is not going to be accessed by multiple applications and it is just an operational database and not being used as a system of record for information. Perhaps the system is a data warehouse with a normalized model and is feeding data down to an operational model that you control. In this case, de-normalization the data, which means storing more data using more database resources, with less relationships and joins can yield simpler SQL and fast access to data with the right indexing. In this approach, one does not need to worry about scheduling of updates to materialized views or joining with fifteen tables with the right keys, aliases, etc. You essentially map the application to how the user is going to access the data to denormalized data structures and the application will scream with performance. Add in a little caching

around content and you have an application that can access data in sub second timeframes.

Let me give you another example. Let us say that an application is being developed that has a web service that manages a user's profile or security access. In this situation, when architecting the platform, it is decided that the web service will provide the granular credentials around permissions around what a user can do in the application beyond the login credentials. Login and permissions are by far the slowness and least scalable part of an IT platform if design and architecture considerations are not made upfront. First, user's credentials and permissions need to reside in memory encrypted in a course fashion. In other words, a big serialized object or linked list or fast collection (ConcurrentHashmap) with permissions. But what if the platform has 10 million customers? If that is the case, will not it mean thousands of services with an infinite amount of memory? The best design pattern in this situation is to pre-load the cache in memory at component startup. You keep all users that have logged in within the last nighty days or that are a new user. This keeps the application screaming fast and only going to the backend database and trans versing data in the event of customers that do not use the application very frequently. It keeps new customers and customers that regularly use the application logging in in under one second vs. waiting ten or fifteen seconds. Remember to achieve great performance, teams must balance around the importance of performance and the priorities. *Figure 2.3* depicts this balance:

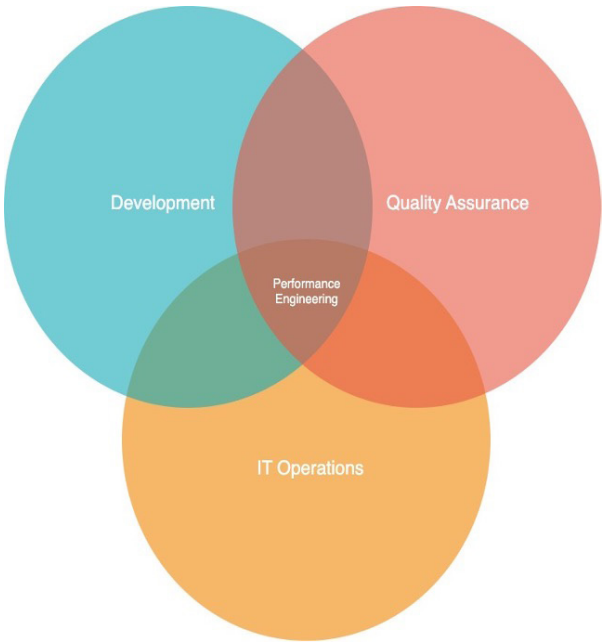


Figure 2.3: *Performance engineering is at the center*

One of the most important groups in an organization is IT operations because they generally put processes in place to assure traceability around changes and enforce rigor into the environment. When the cloud first came out in the

were scared. Organizations looking to leverage the cloud were thinking about entirely eliminating their IT operations staff. There was fear for people not wanting to embrace change and everyone was concerned for their jobs thinking this cloud stuff was going to have developers do everything and there would be no other groups. This is exactly how it started out. Developers started trying to do operations. For senior folks that have worked across multiple domains in IT like dev, operations, architecture, performance, database etc. you can guess how well that worked. Though utopia for many managers is having at IT staff that knows all disciplines, a golden unicorn candidate, this is rarely the reality and the result is generally load quality when you hire a generalist over a specialist. Most developers do not really know the intricacies around operations, they know development which included writing code, putting it into containers and deploying it. Finding items like a PID and how many threads are operating on a PID, how threading works in different operating systems, how to setup rules in balancers, the capabilities around reverse proxies, how to setup proper timeouts and understand timeout dependencies across infrastructure or implement caching that the infrastructure inherently at the different layers was foreign. How specs around CPU or processing power contributed to scale and latency was not understand like it was for folks specialized in this field. Most developers did not know networking or capacity tools like SAR or MTR to understand capacity or ping latencies across network switches or use tools like Wireshark to analyze a TCP dump when interacting between cloud and on-premises platforms. Most did not know how to do a core dump for a platform. But most importantly, they did not know how to configure infrastructure properly across the architectural layers to tune it to what was needed for what each IT layer. They did not know how to setup a web server to have great performance, compression, buffers, caching, and to have these solutions override headers to push caching to browsers or different proxy layers etc. They did not know how to leverage HTTP/3 and the many other items that operations folks understood so well. But just like the operations folks were experts in understanding and managing operations, developers were experts in development which operations folks had a limited understanding of. So, the industry was at a cross roads. Development driving with no operational processes and over sight was problematic and generated defects and poor quality at a high rate. The same situation existed on the operations side as far as understanding development. They had limited understanding. Then google came out with a new concept after these challenges started to arise in the industry. They came up with the concept of **site reliability engineering** (**SRE**).

Site reliability engineering

SRE is the new IT operations group in the cloud. It involves is the discipline of creating highly scalable and reliable software. This is the new IT operations group as applications migrate to the cloud. They continually monitor the platform, receive pages and engage the necessary teams around support.

This group is responsible for:

1. **Service level objectives:**

- Establishing response time thresholds and error budgets. SLOs represent the target performance level for your service. They consist of multiple metrics, such as system throughput and response times, ensuring you maintain a high-quality user experience.
- Establishing response time thresholds is a core component of your SLOs. These thresholds delineate the acceptable time limits for system responses. Think in terms of percentiles when establishing these. For example, 95th percentile response time should be perhaps less than200 milliseconds for a service.
- Consider error budgets to quantify the acceptable rate of failure within a given period, balancing reliability with the need for innovation and faster deployment. For example, if we have an SLO of 99.9% uptime, our error budget may be 0.1% downtime. This buffer creates room for risk-taking and experimentation, which are integral to continuous improvement and innovation in Agile environments. Remember that 99.9999% sounds great but it leaves you little room for innovation and experimentation.

2. **Managing Infrastructure:**
    - SREs play a crucial role in creating and managing infrastructure to support scalable and reliable IT platforms. Most SREs will adopt **infrastructure as code** (**IaC**) to automate the setup and maintenance of IT infrastructure. This approach allows SREs to automate the management of resources like servers, databases, and networks much more efficiently and consistently using **`.yaml`** or other runtime description files.
    - SREs can ensure that infrastructure changes are repeatable and version-controlled using popular IaC tools. These tools help in defining resources declaratively, enabling SREs to provision and manage environments rapidly and with minimal risk of human error. As of the writing of this book the most popular tools are:
        - Terraform
        - AWS CloudFormation
        - Azure Resource Manager
        - Google Cloud Deployment Manager
        - Pulumi
        - Ansible
        - Chef
        - Puppet
        - Crosspane

- Vagrant
- Saltstack
- Spacelift
- Checkov
- Infracost
- Env0

- Moreover, SREs implement monitoring and logging solutions to maintain visibility into system health and performance. Tools like Prometheus for monitoring and **Elasticsearch, Logstash, and Kibana** (**ELK**) stack for logging allow them to detect anomalies and respond to issues promptly. This proactive approach helps in maintaining high availability and reliability of services, aligning with the core tenets of the Agile Manifesto.

3. **Monitoring/Alerting:**

- Dashboards and paging alerts, synthetic and real user monitoring. SREs implement monitoring and logging solutions to maintain visibility into system health and performance. They generally create synthetic monitoring as well as real-time monitoring around what customers are experiencing. These solutions include building custom monitoring and dashboarding solutions or leveraging popular industry tools like OverOps, OpenTelemetry, Datadog, Dynatrace, Prometheus, etc.

- Some organizations leverage the ELK stack for logging allow them to detect anomalies and respond to issues promptly. Regardless of what is used, it is important to maintain high availability and reliability of services, aligning with the core tenets of the Agile manifesto. For tracking over a long period of time, Grafana and time series databases such as InfluxDB are powerful combinations for dashboarding and recording performance for an IT platform.

4. **Capacity planning:**

- Capacity planning is an important task for SREs. To ensure systems run smoothly under varying loads, SREs analyze historical data and use predictive modeling to forecast future resource needs and work with performance engineers to make sure workload traffic models used in performance testing reflect what is in production. They leverage a combination of toolsets and techniques, including short and long-term trend analysis, load testing, and performance benchmarks. By continually monitoring resource utilization and adjusting capacities, they can preemptively address potential bottlenecks before they affect system performance and impact customers. This strategic approach not only enhances system reliability but also aligns

perfectly with the Agile Manifesto's principles of continuous improvement and high quality in development processes.

5. **Resiliency and availability:**

   Running regular resiliency tests is another key responsibility for SREs as well as disaster recovery exercises. These tests, that should include chaos engineering, help ensure systems can handle unexpected failures and continue to operate smoothly. SREs employ a variety of tools to introduce controlled failures into the system, such as shutting down servers or simulating network latency. By doing so, they can observe how the system responds and identify weaknesses that need addressing. This proactive strategy aligns perfectly with the Agile principle of regular reflection and adjustment, promoting robust, dependable systems that can adapt to real-world challenges.

6. **Incident management:**

   - Respond promptly to service disruptions and outages immediately and perform post mortems around root cause around platform issues and drive continual improvement around the IT Platform resiliency. These folks manage and follow the incident process in an organization through to the end.

7. **Risk assessments:**

   - Perform **Failure Mode and Effect Analysis** (**FMEA**) and perform blameless post mortars to understand root cause of incidents.

8. **Automation:**

   - Automate software deployments and implement the needed quality gating around performance and quality.

You see a lot of hype on the industry around DevOps, DevSecOps, and QEOps. These are essentially pairing up the overlap between development and SRE or quality engineering and SRE or the overlap between security and SRE. These clearly highlight the relationships across the different domains and how interrelated they have become with today's automated cloud environment. For on-premises applications, typically you would have an operations group, a development group and a quality engineering group. You would also have groups related to capacity planning, performance testing, usability, security, etcetera. All of these different IT groups would have well isolated sets of responsibilities with clear lines drawn regarding who does what and almost no overlap. The development team would write the software and design the platform. Generally, architects would be a part of this group or their own separate group depending on the organization and how many different products they may architect solutions for. We have seen some architecture groups be in an ivory tower and not understand the details or implementation and

others embedded in teams at the application architecture level. This is not really a good idea and the best results are when architects have a deeper understanding. The quality engineering group would test the software to make sure functional requirements were met and generally track a traceability matrix to make sure that the requirements, taped to the design, which mapped to the code and then testing would determine if these all aligned and each group met the requirements. They would not only determine if the software worked, but if it worked to the specified requirements. Performance engineers and testers are in their own separate group would identify bottlenecks and optimize the platform. Capacity planners would monitor the infrastructure and performance testers would script and load test the application manually before a release.

As software has become more automated, these isolated senses of responsibilities have begun to overlap further depending on how the organization emerged into the cloud space. No doubt over the past ten years organizations have been in transition looking for what works best for them and still trying to understand some sense of the right way to do things. Some organizations indicate that IT Operations in the cloud environment is now called SRE. Google pushed this concept early when the clouds were forming even though Amazon and Microsoft were far ahead of them in the space. SRE's manage and monitor capacity planning, performance, and infrastructure and automate the releases in many organizations or delegate this out and partner with a DevOps group (the right way) that does this depending on the organization. Out of all roles in an organization an SRE is the one most varied when it comes to understanding and responsibility. It became so confusing that Google had to write multiple books on the topic to try to straighten it out to help organizations move from older operations teams to newer organization structures supporting the cloud and automation. At the end of the day however, many organizations differ on their implementation of SRE. In reality, these folks should be all about availability, resiliency, performance, capacity, creating single panes of glass for alerting, observability and DevOps groups should be the groups deploying or automating the deployments. DevOps groups bridge the gap between development and operations. This team plays an essential role in optimizing software delivery pipelines, ensuring a smooth and efficient path from code development to deployment. **Continuous integration and continuous delivery** (**CI/CD**) are at the heart of DevOps practice, allowing for frequent testing and quicker release cycles. Other organizations I have seen push more towards development doing too much to speed releases and do almost everything but at the expense of developers not really knowing operations which leads to unreliable platforms. These organizations are generally less successful because they do not understand all of the features around reverse proxies, balancers, domain name servers, and the various network appliances. Concepts like **time to live** (**TTL**) or keep alive and the abilities around what can be done at the different layers of infrastructure is foreign to them. The result is slower platforms that run at higher cost with less resilience. Taking a SRE approach is a much more reliable approach to building robust platforms and monitoring them properly.

SRE assure appropriate handling of system failures, ensuring a seamless user experience, making systems scalable, while providing system optimization. The value that this

team provides is increased website efficiency, user satisfaction, prevention of revenue losses from system downtimes, and insuring business continuity. Many performance engineering disciplines have been migrated from performance engineering teams to SRE teams as well as chaos engineering which encapsulates resiliency testing and disaster recovery management and execution. This trend has been prevalent in the industry where an SRE does the IT platform optimization, web performance analysis and capacity planning, and resiliency testing for the platform. Then a separate group does performance testing and partners with the SRE team for optimization. From my experience, separation and well-defined responsibilities, ownership and accountability mean non ambiguous role definitions and the proper team charters. Let us talk about a process that assures great performance and when in the product life cycle performance should be measured. Remember that the farther left we measure and optimize performance, the cheaper it is to fix. It means less technical debt in the form of backlog items and allows for optimization in most cases from the developer right when the code is being developed. We always want to shoot for measuring and optimization as early as possible and then measure it throughout the life cycle.

Performance first product development process

Performance first product development process is depicted in the following figure:



***Figure 2.4**: Performance first product development process*

Referring to *Figure 2.4*, the numbers represent areas in the process where performance is a large consideration in the software development lifecycle.

- **Requirements matter**: Define good performance requirements up front. This should be based on competitors' product performance. When a competing product does not exist, then the performance should exceed the State of the Web report listed on the **https://httparchive.org/** website for the top 1000 websites on the internet.
- **Design to performance requirements upfront**: This means taking into account network times and payload sizes for different bandwidths and the number of requests going across the network in the design process across services and the database. Understanding failover latencies and how timeouts across layers impact the scale of the application. Design architecture using a standard design paradigm and leverage best practices around resiliency and performance such as coarse-grained requests and localized caching with event patterns for dirty caches. Minimize requests across the wire. Only use enterprise caches such as Redis where requests go across the wire as common enterprise caches. The platform should be able to not be impacted by a container or pod failing. Execute catered performance tests to assure that the architecture scales as expected. Consider layers in the architecture and TTL caching at each layer. Consider setting standard around retries to keep from overloading the infrastructure on failures.
- **Use PDD**: Measure performance across methods as the code is created. Use best practices to assure that all individual methods are less than one hundred milliseconds as a rule and web client methods calling backend web services across the network return within 500ms. Use a tool with low overhead to measure performance such as the JMH.
- **Continuous testing**: This should occur and as part of the build process for quality gating around performance in the QA environment. Ideally a dedicated performance environment should exist in parallel to the environment used for functional testing or the functional testing environment should match production as far as sizing. If it does not, test with only a few threads to test concurrency and do a load testing in whatever environment is closest matching production as far as sizing. This includes not only the servers, CPUs, memory, balancing and reverse proxy rules, but also the database sizing.
- **Performance environments**: The user acceptance testing environment can also be used for performance testing load if it matches production. Performance testing at some level should occur in all environments (dev unit testing, QA limited thread testing and in pre-prod or a dedicated performance environment matching production sizing.

  In production, we run a quality gate as part of promoting the product into production. We check to make sure the performance is within the SLO and if it is not, we rollback the promotion.

- **Monitoring and observability**: Having end to end visibility into an IT platform is critical to understanding the ongoing workload model for the application which consists of the traffic patterns observed across each architecture layer including web and individual service requests. The percentage of traffic that goes to each endpoint continues to be tested at the same percentages in the lower environment as part of load test scripts that are created.

  Once the software is in production, you want to obtain feedback through surveys or analytics to see if the application is really the expected success. This is extremely important to understand how to drive future requirements for the product. You want to understand if the customer is bouncing from the application where a poor experience exists or sluggishness in the application. The same can be said for functional aspects around routes the user may be taking in the application. If leveraging an **application performance monitoring** (**APM**) tool that saves customer sessions like Datadog, then reviewing the slower sessions is a good idea to understand what the customer experienced.

Performance driven development with JMH

When do we test in the process. We test during design, unit testing during coding using PDD, and heavily in integration testing at a low level of concurrency. Then finally in a pre-production performance environment at full load. Though all of the features of the different java libraries and tools are beyond the scope of this book for PDD, below is an example of how to leverage JMH for each method when unit testing and coding an application. This is the most basic example of using JMH for measuring latency during the development process for each method. Because we did not configure how many threads to use, it will default to twenty warmups and twenty test iterations. This gives us some consistency in the performance measurement minimizing the standard deviation in response time across test runs. JMH is extremely accurate with much less overhead than taking a logging approach to output time using time functions in Java such as **`System.currentTimeMillis()` o**r using heavy weight tools like JMeter or Gatling. It was designed purposefully to not introduce more than two milliseconds of overhead; less than an IO statement writing to the screen in a web application. Here is where you can find more information on JMH including some of the profilers and samplers: **https://github.com/openjdk/jmh.**

```
//Test Program to showcase JMH testing in PDD
import java.io;
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;

import java.util.concurrent.TimeUnit;

 @BenchmarkMode(Mode.Throughput)
```

```
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class BenchmarkMethodExample {

      @Benchmark
      public void benchmarkMethod() {
          System.out.println(“This is a test”);
       }

      public static void main(String[] args){
          org.openjdk.jmh.Main.main(args);
        }
}
```

Load testing APIs with JMH

What about service level API Testing? Here is how to use JMH to load test a web service. Yes, we said load test! It will first warmup the connections and then run a load test against the service running five iterations with one hundred threads. This is a very low overhead way of doing load testing for APIs with minimal response time standard deviation. It has the basics around what is needed without the features and larger deviations around response times seen with most of the load testing products which unfortunately also end up contributing to the latency being measured. The disadvantage of less features is also less features however. When doing simple load testing against APIs which do not require much complexity, it is best to choose lighter weight solutions.

```
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.openjdk.jmh.annotations.*;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@Warmup(iterations = 2, time = 5, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 10, timeUnit = TimeUnit.SECONDS)
@Threads(100)
@Fork(2)
public class BenchmarkRestAPI {

       private static final String API_URL = "your app url";
      @Benchmark
      public void restApiRequest() throws IOException {
```

```
            try (CloseableHttpClient httpCl = HttpClients.
createDefault()) {
                HttpGet httpGet = new HttpGet(API_URL);
                HttpResponse response = httpCl.execute(httpGet);
                HttpEntity entity = response.getEntity();
            }
        }

        public static void main(String[] args) throws Exception {
            org.openjdk.jmh.Main.main(args);
        }
    }
```

Performance journey level automation

Detailed Java implementations using tools like JMeter or Gatling is also out of the scope of this book. However, below is a simple example of a JMeter application written in Java to measure the time to login to website for reference. You do not have to use Java source code when working with JMeter, but when it comes to automation instead of going through the user interface of the JMeter application, it is very easy to program a nice test that is very automatable using the Java language as shown below. Essentially, the pattern is always the same, you create a loop controller, sampler, and the threads and then just run the engine to initiate tests. More information about JMeter can be found at **https://jmeter.apache.org/**.

```
import org.apache.jmeter.control.LoopController;
import org.apache.jmeter.engine.StandardJMeterEngine;
import org.apache.jmeter.protocol.http.config.gui.HttpDefaultsGui;
import org.apache.jmeter.protocol.http.control.CookieManager;
import org.apache.jmeter.protocol.http.control.HeaderManager;
import org.apache.jmeter.protocol.http.sampler.HTTPSampler;
import org.apache.jmeter.testelement.TestElement;
import org.apache.jmeter.testelement.TestPlan;
import org.apache.jmeter.threads.SetupThreadGroup;
import org.apache.jmeter.util.JMeterUtils;
import org.apache.jmeter.util.SSLManager;
import org.apache.jmeter.util.TestPlanGui;
import org.apache.jorphan.collections.HashTree;

public class JMeterWebLoginTest {

    public static void main(String[] args) {
    try{
```

```
            HashTree testPlanTaxonomy = new HashTree();
            TestPlan testPlan = new TestPlan();
            testPlanTaxonomy.add(testPlan);
            HttpDefaultsGui httpConfig = new HttpDefaultsGui();
            testPlanTree.getTree(testPlan).add(httpConfig);
            LoopController loopController = new LoopController();
            loopController.setLoops(1);
            SetupThreadGroup threadGroup = new SetupThreadGroup();
            threadGroup.setNumThreads(1); //concurrency
            threadGroup.setRampUp(1); //ramp up amount
            threadGroup.setSamplerController(loopController);
            testPlanTaxonomy.getTree(testPlan).add(threadGroup);
            HTTPSampler loginSampler = new HTTPSampler();
            loginSampler.setDomain("yourdomaingoeshere.com");
            loginSampler.setPort(80);
            loginSampler.setPath("/loginpathtologinuser);
            loginSampler.setMethod("POST");
            loginSampler.addArgument("username", "your_username");
            loginSampler.addArgument("password", "your_password");
            testPlanTaxonomy.getTree(threadGroup).add(loginSampler);
            CookieManager cookieManager = new CookieManager();
            cookieManager.setName("Cookie Manager");
            testPlanTree.getTree(threadGroup).add(cookieManager);
            HeaderManager headerManager = new HeaderManager();
            headerManager.setName("Header Manager");
            testPlanTaxonomy.getTree(threadGroup).add(headerManager);
            StandardJMeterEngine jmeter = new StandardJMeterEngine();
            SSLManager sslManager = SSLManager.getInstance();
            sslManager.setContext(jmeter.getDefaultSSLContext());
            jmeter.configure(testPlanTaxonomy);
            jmeter.run();
        }catch(Exception e){
                System.out.println(e.printStackTrace());
        }
    }
}
```

Then, because JMeter does not measure JavaScript or any frontend times which is half of the equation when it comes to performance engineering, you need to also leverage Selenium or other frontend tool or script that can pull out attributes like **Largest Contentful Paint** (**LCP**

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.devtools.DevTools;
import org.openqa.selenium.devtools.v98.performance.Performance;

public class WebPageLCPMeasurement {
    public static void main(String[] args) {
        System.setProperty("webdriver.chrome.driver", "/path/to/
chromedriver");
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--headless");
        WebDriver driver = new ChromeDriver(options);
        DevTools devTools = (DevTools) driver;
        devTools.createSession();
        devTools.send(Performance.enable());
        driver.get("https://yourwebpage”);
        try {
            Thread.sleep(5000); // Sleep for 5 seconds (adjust as needed)
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Retrieve LCP value from performance metrics
        WebElement lcpElement = driver.findElement(By.cssSelector("your-lcp-
element-selector"));
        double lcpValue = Double.parseDouble(lcpElement.getAttribute("data-
timestamp"));
        System.out.
println("LCP (Largest Contentful Paint): " + lcpValue + " milliseconds");
        driver.quit();
    }
}
```

Leveraging JMH for testing performance in Unit Testing as part of PDD, then JMH for load testing APIs. Following up with journey level performance testing across frontend web applications using JMeter combined with Selenium is the key to full end to end performance testing. You need to be able to test end to end through the UI and understand latency at every step and have those results aggregated to a transaction related view ideally or user feature or journey. Of course, other tools are available like LoadRunner, Gatling, etc. But remember at the end of the day you need to be able to measure at the most granular level

all the way up to the user interface level and have quality gates around each level. Being able to fully automate performance against quality gates with SLO thresholds is what you are after. You want to fail builds when going from one environment to the other and you want to run continues automated testing so you know the state of performance over time and the trends.

Items to always consider before testing:

- Identifying the testing environment as a first step. You need to know the hardware, compute power, memory, disk latency and disk type, software and network configuration. Have a full infrastructure and architecture diagram available and trace the application end to end. Never test an existing application without this information. This is crucial to understanding what is being tested. Configure the hardware to reflect the number of requests coming into the platform and consider the needed throughput before ever initiating a performance test. Many cloud enviornments have different pay tiers around disk, compute and memory speeds. It is important to understand the needs of the platform being built.
- Understand how users interact with your application to emulate those same interactions in your tests.
- Set goals for response times, throughput, and resource allocation and error rates.
- Utilize the user behavior data and expected load patterns to put your application under test.
- As tests initially are created for the first time prior to automation, carefully monitor and analyze the results to ascertain where and why errors are happening.

Defining the workload model

When you have done all of the pretesting work, it is time to create the workload model. The workload model is a crucial aspect of the performance testing process. It represents a set of user behaviors that your application or website is likely to encounter during its operation. By simulating these behaviors, you can effectively evaluate how your platform will perform under different traffic patterns. The important thing to remember is that you need to create a workload model that does not just manage the expected happy path scenarios, but also negative scenarios at the expected negative rate with test data. One of the worst things that an application can do is not have proper error checking. A single exception throws is equivalent to a hundred lines of code execution on average as it throws and creates the call stack trace. It should truly be an exceptional operation that should only occur when there is something that happens infrequently in the application. When you start throwing exceptions for everything that are not really errors and expected behaviors, the performance of the platform will suffer greatly. The JIT compiler is unable to optimize exceptions well which generally means these are interpreted statements. When you do not test these scenarios properly, you are giving yourself very

with the results being much better than what would be experienced in the real world. Creating an accurate workload model is a complex task. The process typically follows these steps:

1. Collect production metrics and gather data about user activity of the platform including traffic patterns, exception types, rates, data, and response time information. Also, metrics like CPU, Memory, Java threads used at peak traffic periods. Understand the patterns for new users coming onto the platform if they differ from what is in production today and also if it is a brand-new platform.
2. Use analytics tools to identify the common patterns of user behavior on your application. This might include the major paths in the application or searching, login/logout actions, data entry, etc.
3. Break down these patterns into individual operations or processes. This step allows you to test each operation independently for more detailed results.
4. Combine various operations to create user scenarios that reflect realistic usage of your application.
5. Simulate these user scenarios under various loads to see how your application performs under different levels of user activity.
6. Remember, establishing an effective workload model is part art and part science. It requires a deep understanding of your platform's operations and a keen intuition about how users are likely to interact with it.

Conclusion

We discussed a great deal about process in this chapter. You should now understand the different process types and where performance fits into each process. The project iron triangle was discussed in detail as an import concept in product management. You will be able to understand the different approaches to product delivery and how performance plays a role in the product creation and why it matters. Culture is discussed along with Agile and DevOps concepts around product delivery. You now understand some of the project challenges around the different methodologies and where performance plays an important role.

In the next chapter, we will discuss the challenges and realities around cloud performance which includes explaining the differences between cloud and on-premises platforms with respect to capacity and responsiveness considerations. We will talk about obstacles for performance engineers and software developers when it comes to scaling complex distributed architectures. **Event drive architecture** (**EDA**) and micro-services will be discussed and the advantages that they play in high performance platforms. chaos engineering will be introduced and considering resilience while still having great performance.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 3

Challenges and Realities of Cloud Performance

Introduction

This chapter of the book will introduce you to some of the challenges and realities around trying to achieve good performance for large distributed applications that operate in the cloud. We will discuss the advantages of cloud versus on-premises solutions and talk about resilience while achieving great performance. We will also discuss the fascinating topic of performance uncertainties and how to calculate the accuracy of results being presented to the stakeholders.

Structure

This chapter covers the following topics:

- On-premises versus cloud IT platform development considerations
- Cloud performance challenges
- Uncertainty of measurement
- Navigating the challenges between on-premises versus cloud optimization
- Complex distributed cloud application evolution
- Event-driven architecture

- Microservices
- Chaos engineering

Objectives

By the end of this chapter, you will be able to understand the benefits of developing new IT platforms in the cloud and when on-premises solutions should be considered. We will discuss the advantages of each and compare the different approaches. We will also discuss chaos engineering, **event driven architectures** (**EDA**) and micro-services and the roles they play in performance and scale for cloud platforms.

On-premises versus cloud IT platform development considerations

Understanding the reasons behind choosing on-premises platforms versus a cloud solution can help guide your decision-making process. No doubt that using on-premises platforms gives us more control around the architecture and performance of IT platforms. Especially if you have a limited product set, this is a great approach to keep the costs low. The cloud lends itself to rapidly creating new products very quickly which is a large advantage but let us assume you are a restaurant that has a single e-commerce site or a company with a small set of applications, on-premises platforms may be the best option in a longer term to keep runtime costs controllable. On-premises platforms allow you to make a one-time investment for long-term use and leverage less overhead that impacts performance. For example, let us say that you are a popular restaurant chain and you have a great web application that helps consumers order pizza. This, and perhaps a couple of other back-office applications is all you have to manage the business. The traffic coming into the platform is only fifty requests per second used to take orders and manage them. In this situation, you have to ask yourself if using an on-premise platform that you have full control over that can run long term at low cost may be a better option than cloud solutions.

This financial model itself is often more predictable than other development models, such as pay-as-you-go cloud services where the rates can fluctuate or increase significantly at a moment's notice and where completion is limited across only a few providers. When you have predictable scale, this also simplifies things. If you go the bare metal route for running applications on-premises, the performance is even much better than virtualized solutions when working with packaged containers. This means that you have more horsepower, in the neighborhood of forty percent more on average, between packaging solutions like using docker running under a virtualized environment versus bare metal alternatives. Running on bare metal with the lowest amount of operational layers translates to low cost being able to run a low throughput predictable scale application with less hardware.

Other advantages of using on-premises platforms are as follows:

- **No dependency on the internet**: Working with on-premises platforms, you do need not rely on internet speeds or costs which can often fluctuate; thus, it offers stable budgeting.
- **Physical security and access**: Since your server infrastructure is on your premises, you can implement stringent security measures and have the direct physical access to troubleshoot issues. This helps in avoiding outsourced IT costs.

On-premises platforms encourage optimized platforms:

- **Customization**: On-premises platforms allow flexibility and control over the server configuration. This encouragement to tailor-make your infrastructure leads to performance optimization and better architectural choices.
- **Faster response times**: Since the servers are located within your premises, data transfer often happens at a much quicker rate than cloud services.
- **Quality assurance**: Having control over the physical servers means you can conduct comprehensive testing activities in real-time, ensuring good architecture and performance practices.

Adopting on-premises platforms is not merely a preliminary decision. It is a strategic measure that helps you maintain cost control and encourages you to adopt optimal practices to develop a robust IT platform.

Cloud platform advantages

When considering the current technological landscape for building IT platforms, why should we choose software development in the cloud? Cloud development holds a variety of advantages that serve to ensure the efficiency and effectiveness of software development projects. Though many may tout a large list of advantages around using the cloud for development based on features available, the largest advantage in a nutshell is rapid IT product creation in a fraction of the time of on-premises solutions. **Site reliability engineering** (**SRE**) teams and architects can design/build solutions and spin up new infrastructure including databases quickly to support a great deal of scale when performance is considered upfront. That being said, cloud runtime costs can become higher over time and be unpredictable and due to the simplicity around creation many times the same level of thought is not put into the solution as with on-premises platforms. The mindset can be that the cloud provider should just take care of everything where solutions should just be able to be quickly wired together and work. Cloud solutions unfortunately lend themselves to running low quality software due to developer perceptions and expectations around building IT solutions in the cloud. It is important to put the necessary life cycle processes in place to assure that this does not happen and evangelize the importance of performance and scale when choosing a cloud environment for development. One should not need to spin up fifty virtual machines or fifty pods in a Kubernetes cluster (container management runtime platform) for an IT platform processing at a rate of ten requests per second for most real time web applications.

Junior developers working on cloud software can feel that scale is not really a concern like it was with on-premises solutions, because they believe the cloud has infinite scale. Thus, they end up writing inefficient code which translates to much larger runtime costs over time. It can sometimes be faster to write software without refactoring or optimizing code to get it out the door quickly. Unlike on-premises platforms, where code needs to be very efficient or we run the risk of purchasing new hardware. This can be costly and also slow down delivery for an IT platform if a server is not already in-house. What kind of behaviors are common? When developers are working in a high velocity IT product creation environment, they want to get something out the door quickly. But, instead of modifying existing code and refactoring the solution each iteration in the development cycle to optimize it, junior resources may just copy the code entirely and modify the new copy for their new feature and do not consider the performance impacts at all; not to mention the maintenance costs for the product.

Un-optimized and non-refactored code replication quickly becomes a common development pattern which can permeate like a cancer if left un-checked. Slow code in one place becomes slow code in another and more code overall means more resources being consumed. The thought process is that it is easy to just spin up another container when the code does not scale or copy code from one service to another and not consider efficiencies. This behavior doubles the code size and slows the platform down but if the perception is that the platform does not need to be efficient because the cloud just takes care of it, this mindset gets an organization into trouble. The same problems occur when it comes to good resiliency.

If there is a perception that the cloud is self-healing, then it can mean developers may not put in the needed error handling or resilience in the code to prevent processes from failing and restarting, which ultimately provides a poor experience for the customer and long latencies impacting scale. Organizations may not run the needed disaster recovery exercises across regions either, thinking that the cloud just takes care of everything. This again, translates to higher runtime costs and a poor customer experience. This also translates to revenue risk.

The largest advantage of using cloud technologies for building IT platforms is for companies that create a lot of different products and have many projects in the pipeline. Developing in the cloud means that you can rapidly create new IT products to sell. Using cloud solutions, you can grab managed services that are already available to provide you with features to use without much effort.

Let us explore these advantages for businesses:

- **Scalability**: Cloud-based services are ideal for businesses with growing or fluctuating bandwidth demands. If your needs increase, it is easy to scale up your cloud capacity, drawing on the service's remote servers. Likewise, if you need to scale down again, the flexibility is there with cloud services. In most cases, scaling can occur based on traffic patterns or time of day.

- **Disaster recovery**: Businesses of all sizes should invest in robust disaster recovery, but for smaller businesses that lack the required cash and expertise, this is often more an ideal than reality. Cloud is now helping more organizations buck that trend. It gives them the ability to operate in multiple regions so if a disaster would occur, they could still do business. Something that was not attainable with an on-premises platforms without significant expense.
- **Cost-effective**: With cloud, there is no need to spend huge amounts of money on purchasing and maintaining equipment. This significantly reduces the capital expenditure and gives you access to a vast amount of computing power you would otherwise not have access to. Again, it depends on the company and if new products are being spun up at a rapid frequency. This is the best case for cloud IT platforms.
- **Collaboration efficiency**: Collaboration in a cloud environment gives your business the ability to communicate and share more easily outside of the traditional methods. This makes the life of developers a lot easier. When the entire development environment is put in the cloud, organizations can see some great efficiencies around communication and development. No installs of bulky **integrated development environments** (**IDE**).
- **Automatic software updates**: In a cloud computing environment, servers are off-premise, out of sight and out of your hair completely. Suppliers take care of them for you and roll out regular software updates, including security updates. Sometimes this is not completely seamless where organizations may need to rebuild their software with updated containers. But compared to on-premises platforms trying to install updates on web servers, application servers, reverse proxies, and balancers, working in the cloud can be a godsend.

Cloud performance challenges

After being involved in designing and developing distributed platforms for so many years, when the cloud came into the picture, promising infinite scale and self-healing services, it was intriguing to all of us. However, thinking about all world organizations everywhere putting their infrastructure into the hands of just three major cloud providers, many with the same political beliefs, may give us some pause. It makes one think about what could potentially go wrong in this type of scenario with all of the organizations of the world putting their infrastructure and IT platforms managed by a small few. More competition and diversity around affiliations would be our preference. But after many years now with *Microsoft*, *Google*, and *Amazon* taking the majority of the market and obtaining revenue more than **gross domestic product** (**GDP**) of a small country, what we have seen is platforms that are designed with low quality at the expense of paying much more for runtime cost, support costs and failures. What do we mean by this, is not running platforms in the cloud cheaper? The advantages of the cloud have been the ability to put something together

quickly as far as feature development for software. Some organizations even have an IDE in the cloud, which allows them to develop, build, run, deploy, and do quality and performance gating in a simple execution step. The cloud shines when having a product concept and needing to build it fast. Most recently, if you have been following American politics, you know that the **Department of Justice (DOJ)**, is asking for *Google* to be broken up and is looking at all of the tech giants. How does this impact what is being created in the cloud, costs, resiliency?

The problem sometimes associated with high-velocity delivery is letting more technical debt into products and letting that accumulate over time, which continues to increase runtime costs and decreases customer satisfaction over time, impacting an organization's brand. The same time slices in the process around reviewing the rigor for an application is less than when products go through a longer development cycle. Automation is critical for building quality gates into processes and even though Agile is all about less documentation, formal signoff processes need to exist in the process at stages in the product's development life cycle and more rigor around the backend runtime, resilience, performance, and functional reliance around negative scenarios needs to be fully vetted. When building an application that has a six-month window, there are a series of meetings, formal code reviews with multiple developers and many eyes on the application. What if scenarios are fully vetted as part of the process. Rigor around operational processes occurs, and putting formal procedures in place to support the application. Documents are reviewed over a period of time for anything that is missed. Eyes are on the effort more and more discussions take place.

Going fast can be a disadvantage when it comes to rigor and considerations around support because it is hard to fathom that if you can put together an application in 12 weeks, that you need at least another 12 weeks to execute what is necessary to put rigor into running the product to run it long term. This includes building monitoring dashboards, code and infrastructure optimization and failover across regions for databases, components, etc., performance quality gating and automation, alerting strategies, support training, functional quality gating and automated regression testing, resiliency testing and much more. Beyond the fast build of a product, making the product have the ability to run at low cost is a very large part of the equation. Though labor costs do reduce generally for building infrastructure and maintaining it, monthly expenses for actually running the software may be more than on-premises solutions depending on the quality of the build and support needed. For example, did the application do unnecessary amounts of overly verbose logging, does the application fail with exceptions often and not have the needed rigor or checks and balances due to rapid delivery. Are try catch blocks in the source code and error handling setup for only true exceptional cases versus controlling program flow slowing down the application. Has the database been properly sharded and partitioned correctly with the right indexing and key relationships. Does the application have the proper timeouts set across operational layers, and does the application still meet SLO when the platform is degraded? Has performance automation been put in place to address performance challenges in a short product delivery window that may be delivering

features each week. Keeping infrastructure up to date has been much simpler in the cloud and the good news is that security issues are addressed sooner than typically with on-premises solutions.

The key point to remember is to always build rigor into any platform that is created and do not let rapid product development translate into poor resilience and scale which unfortunately has happened in many organizations in an effort to get a product out the door to sell it. Remember that though the cloud has allowed different products with different architectures to be created quickly, building the product is only half of the equation if not less. The processes and rigor for resiliency and ongoing support to keep ongoing runtime costs low is very important. If you do not do this it means an expensive platform that is hard to maintain with customer dissatisfaction impacting your organizations brand.

Ultimately, cloud providers did not fulfill their promise around infinity scale, simplicity around performance and self-healing, at least not yet, but it does have some promising dynamics and it is still evolving. What we do see is cloud providers driving architectures to benefit their own revenue streams like all good businesses would, but the side effect encourages some poor design and monitoring practices that eat up resource utilization which benefits the provider. Perhaps **artificial intelligence** (**AI**) will be the catalyst that changes that, but instead, it will more likely be a great revenue stream opportunity to grab more computing time. Compartmentalizing cost structure by compute was ingenious as a revenue model and one that continues to drive more features that take up more compute resources. They did successfully get organizations onto their platforms and it did not take long. Why would organizations do this?

As a young boy in third grade, a lesson was taught to me regarding how to advertise. It was called the bandwagon philosophy which meant that if everyone else was doing it, perhaps a bunch of somebodies, it must be the right thing to do. But that is an over simplistic explanation and not entirely truthful when we talk about the reasons for organizational cloud migration. It is important to remember that organizations are competing for market share and competing around products being created for IT platforms. Yes, sometimes that means everyone is jumping on the bandwagon. Organizations develop software to automate something from a process perspective or to sell something to generate revenue and having high performance and great features is a large part of that. The cloud though, wo not necessarily save on runtime and support over the longer term, in fact it many times facilitates the creation of bad coding practices and delivery because one can always on the fly ask for more CPU, memory, pods, nodes, etc. and it becomes more about speed than quality. We talked about this earlier. As with all sales pitches around the cloud, over simplification many times leaves out the needed details to make something work. Cloud computing has without a doubt transformed the world of technology, bringing forth vast efficiency and scalability benefits. But, as with all new innovation, it brings its own set of challenges; particularly in realizing optimal performance in complex distributed cloud applications. This book is dedicated to helping you navigate through these challenges which often revolve around scaling, performance and the use of micro-services with EDA. Do these challenges sound familiar? Are you feeling a

we plan to untangle some of these complexities and provide a path for you to understand and improve your cloud applications' performance. With the right understanding, optimization strategies and tools, navigating the difficult art of cloud performance can become much less intimidating.

Cloud managed services

Most cloud providers today provide an ability to speed up creation of cloud platforms using managed services. Using these depends on an organization's needs. Managed services refer to the practice of outsourcing the responsibility for maintaining, and anticipating the need for, various processes and functions in order to improve operations and cut expenses. This is particularly relevant when it comes to cloud IT platforms. Many times, this even comes in the form of black box services that cloud providers offer. These are easy to create and scale up without the complexity of container managed systems like Kubernetes. Software can be created and deployed in far less time that is easier to maintain and less costly. But the downside is the ability to optimize these black boxes are very limited and they eliminate any possibility at all of moving a platform from one cloud provider to another easily.

Why do businesses lean towards managed services in the cloud? Primarily, it is about efficiency and expertise. Managed services provide access to black box services or specialized skills and technologies that you might not possess internally, allowing you to focus on your core business activities rather than IT maintenance and troubleshooting. If the team does not know docker or Kubernetes, a cloud provider may have a black box container manager that simplifies the whole process, saving hours of setup time and support costs.

Additionally, managed services offer scalability out of the box. Cloud platforms can grow with your business requirements without the need for investing in extra resources or other software. This means you can adapt to meet client demands quickly and scale up efficiently.

Cost management is another reason. By leveraging black box services or third party managed services, organizations can often achieve a more predictable expense structure. This makes budgeting more straightforward.

Managed services ensure higher availability and reliability. Cloud providers typically guarantee uptime and performance standards, which translates into smooth and continuous service delivery.

How different cloud components impact performance

When considering cloud platform performance, several components come into play. Network Latency tops the list, as data must travel between user devices and cloud servers.

Consider the size of the data being transferred across the network and the number of hops and thus potential for slowdowns with standard deviations around response times due to tolerances and traffic going through different network devices. Any request over 5kbytes should be compressed, connections regularly used should be kept alive for performance reasons.

Data that marshals from one format to another or that needs to transform itself into different object graphs or structures to be consumed can become bottlenecks for an IT platform. High latency, due to expensive operations with respect to CPU consumption from converting information from one format to another, can significantly slow down operations. From a design perspective, the goal should be to not marshal between formats across distributed tiers any more than necessary. For example, for a web application, if you use a NoSQL database that stores data in JSON that sends data directly back to a browser that understands JSON with no parsing or marshaling in between, is going to be extremely fast and responsive. Bringing back data from a relational database, marshaling that data into an object graph, converting that to objects to be transferred in another object to go across the network, and then marshaling that data then into a JSON or XML format before being sent back to the browser, can be a very slow design. If using a lot of marshaling or parsing, better to not have an entire domain object graph in memory and instead directly go after the data being requested and have an object graph of the resultset versus different objects representing different tables or objects. This minimizes the transversal and formatting and only works on what is needed for the specific request.

Next, think about compute resources: The virtual CPUs and memory. Insufficient resources can lead to bottlenecks, affecting application speed and responsiveness. Similarly, storage performance plays a crucial role. Accessing data from slow disks can lead to delays, so it is important to ensure that storage is optimized. Most cloud providers will support different tiers of performance for disk latency. The higher the tier, the faster the performance. For example, if the IT platform does a lot of logging to disk, the higher tier with faster disk writes can make a world of difference when it comes to performance and scale. If the platform is logging a lot of information for audit purposes, the choices are to use fast disks or use a buffer to speed up threads writing to disk and releasing them sooner to support more incoming requests or additional logging. Many logging frameworks have this ability-built in. Remember that if you use fast disks, the platform will be fast in general because Linux, Windows, and other operating systems swap to disk as the platform operates. The faster that swap happens, the faster the operating system supporting the application will run.

Additionally, the scalability of the infrastructure is vital. As demand fluctuates, systems must automatically adjust to provide additional resources when necessary. Load balancing ensures that incoming network traffic is evenly distributed across the server pool, preventing any one server from becoming overwhelmed. Furthermore, software design, including how applications are architected and coded, can greatly impact performance. Optimize algorithms and ensure efficient coding practices to maintain speed. Beyond code reviews, third party tools like PMD or SonarQube plugins around code quality can make a world of difference.

Lastly, consider how security protocols are integrated. While security is paramount, poorly implemented security measures can introduce latency. It is important to have SSL caching and optimized to balance performance with security. Balancing security with performance requires a holistic approach. Understanding and regularly evaluating these components is crucial for achieving the best possible performance in a cloud platform.

Region to region latency

When discussing region-to-region latency in an IT platform, it is important to consider the geographical distance that data must travel between data centers and any edge caching that needs to be in place at different geo-locations. This distance can result in longer response times, impacting the performance and efficiency of distributed systems. By understanding and simulating the latency effects between different geographic regions, we can address potential bottlenecks and improve our platform's overall performance.

Incorporating chaos engineering experiments can help us measure and mitigate the impact of regional latency on our system. Through these experiments, we can simulate real-world conditions and develop strategies to optimize our system's ability to handle cross-region data transfers. By proactively tackling these latency challenges, we ensure a seamless user experience and fortify our system against disruptions in our IT platform.

Resilience, failover and scale impacts

When we talk about resilience in an IT platform, we are referring to its ability to withstand and recover from disruptions. This involves designing systems to continue operating even under failure conditions. Resilience is essential because it minimizes downtime and ensures the consistent delivery of services. We achieve this by implementing redundancies, such as backup systems and parallel processing, which help maintain functionality during unexpected events.

Now, let us consider failover. This is essentially the system's response mechanism to failures. Failover automatically transfers processing tasks to a standby system in case of a primary system failure. This may be an active system or an inactive system that must be started with manual procedures after alerting. Ideally, the system incorporates an automatic transition to an already active system which is crucial for maintaining service continuity not impacting users. It kicks in to keep everything running smoothly without human intervention. Failover plans need regular testing to ensure they function as expected when required. Remember though, that failover system must be able to handle the traffic load. If the platform is active where one region fails and the other picks it up, that failover region must be able to handle the entire load without impacting defined service level objectives.

Scalability refers to a platform's ability to handle increased loads efficiently. As your IT platform grows, either in data volume or user traffic, it must scale to manage this increased demand. Effective scalability ensures that performance remains consistent without

degradation, accommodating growth seamlessly. Without proper scaling, we might face performance bottlenecks, leading to poor user experiences and potential service outages.

Security impacts on performance

Security impacts our IT platform's performance. Robust security measures ensure data protection and safeguard against breaches, the keys over time for encryption tend to get larger and more complex, but they also add latency and overhead affecting an IT platforms performance. This is important to consider in the platform design and architecture. It is important to design our security architecture in a way that minimizes these impacts and ensures the needed caching and reduction of calls across the network.

Encryption is essential for protecting data integrity and confidentiality, but it will also slow down data processing. One of the largest impacts on CPU time is encryption and decryption.

CPU Time is the duration a processor spends executing a specific thread or process. It is a crucial metric for understanding how efficiently our application runs. By measuring CPU Time, we can identify potential inefficiencies in our code and improve performance, especially in a cloud platform environment where resource optimization is key. Regular monitoring can help prevent CPU failures and ensure smoother operations by anticipating and addressing potential bottlenecks. Reviewing how different key sizes and security algorithms impact CPU time and architecting solutions to minimize impact is the key to developing great responsive products.

Implementing efficient encryption algorithms and optimizing security protocols is key to maintaining performance levels. Security checks, though necessary, will impede system performance and processing speed, so strategically placing these checks and using caching can mitigate performance hits is important.

Beyond encryption, consider implementing other security measures like **digital rights management** (**DRM**) and **Secure Socket Layer** (**SSL**) protocols. These can improve our defenses against unauthorized access. It is important to ensure these layers do not create bottlenecks. It is important to conduct regular reviews of our security protocols to identify any performance issues that could arise from software updates or platform changes. Implementing security designs without considering the impacts on the end-user performance and usability is irresponsible. We must realize a security unusable platform that builds a bad brand experience is not something we want and it ultimately impacts our organizations revenue. For cloud-hosted applications, the complexity multiplies compared to on-premise platforms. Each layer and component of the cloud infrastructure introduces a potential point of failure or performance concern, especially during critical updates that might require downtime. Here lies the importance of chaos engineering. By introducing controlled disruptions, we identify how these layers handle stresses and pinpoint where security protocols might crack under pressure. This proactive approach to resiliency and security is essential for maintaining optimal performance. Ultimately, the balance between

security and performance is not just about choosing the best tools but creating a holistic strategy. This includes integrating robust observability / monitoring systems to detect and resolve issues swiftly, thus minimizing any performance impact. This proactive strategy ensures that our IT platform remains secure and efficient, even as it scales or adapts to new challenges.

Uncertainty of measurement

Cloud performance is never straightforward, thanks to the highly distributed nature of the architecture and the immense complexity involved in managing and optimizing these systems. These systems often include various components such as databases, web servers, application servers, and more. Also, the use of event driven architectures and micro-services bring their own set of performance challenges. The major problems around managing cloud performance are due to the complexity and inability to effectively monitor, analyze, and manage the amounts of data and correlate it when processed through these systems. It is also difficult to understand uncertainties around measurement. Early in my career, Metrology was my focus and automating finite measurements around microwave technology. It meant designing software using the basic computer on an HP9000 or in Pascal on an HP3000 and having it control *IEEE-488* interfaces to control electronic equipment. These platforms were able to run the code completely in memory and performance was exceptional. Performance was never a problem, even though they did not have near the horsepower by today's standards. When the code was run, it remotely controlled the amplitude of signals, managed settings for frequency synthesizers, network and spectrum analyzers and other pieces of expensive military grade electronic equipment. Then sampling the output from measurements to determine of the piece of hardware being calibrated was within the specified tolerance.

Uncertainty is a complex understanding. Remember that uncertainties are additive across layers of a platform where requests are serial in nature and are not when it comes to parallelization. As requests hop from one architectural layer to the next and different components some serial and some parallel, uncertainty of measurement becomes very important. The approach that was taken thirty years ago in metrology is the same approach as taken today when measuring uncertainties across any platform running hardware and software. The same approach is used for everything from electronic circuits to distributed software platforms. It means using the *National Institute of Standards and Technology* (*NIST*) principles which means using the **guide to the expression of uncertainty in measurement** (**GUM**). GUM is a simple formula and the most important one when it comes to good measurements around performance. It is stated as:

$$Uncertainty = k * u$$

- k is the coverage factor that depends on the desired level of confidence
- u is the standard uncertainty of measurement, which is the combination of all components in the system

Parts of the system are broken up into different types of components. Let us call them Type A or Type B components which is a popular way to describe the dynamics. Type A components have an important attribute when it comes to uncertainty, they are able to be evaluated by statistical methods such as standard deviation across multiple measurements. When you measure against SLO in automation, it is very important to take multiple measurements. For example, if you are running at test with tooling such as JMeter or JMH with a web service client for performance measurement under load, you would never want to take less than three measurements because it is very important to understand the standard deviation for a component along with the uncertainty of the tool taking the measurement.

The other type of component that is important to calculating overall uncertainty we will call Type B components. This is where uncertainty around components is identified using methods, such as vendor specifications versus measurement is used. For example, if we are using the *Google Cloud* and *Google States* that they have a service that will operate within 1 second of latency plus or minus 200ms, then we can state that the uncertainty is 200ms and add this into our calculations beyond uncertainties around the custom software we are creating where statistical methods are being used to evaluate them. To calculate the uncertainty around a platform is a simple one. You would take the square root of the sum of the squares around all uncertainty for each component.

Uncertainty = sqrt (u1^2 + u2^2 + ……. other components or vendor written specs)

Hence, if you have a web service that has a standard deviation in response time of 200ms and you have a vendor component that has an uncertainty of 300ms and you are using test software that has a deviation of 100ms, then you would leverage:

Uncertainty = sqrt (200^2 + 300^2 + 100^2)

This is the combined uncertainty of measured and specifications around vendor software and hardware. If you have hardware that varies due to tolerances specified in the electronics, this can be also be included in the calculation. The goal is to combine all measured along with vendor specified uncertainties through third party documentation. This will tell you when you test or present results how to measure the performance properly and how much uncertainty you need to account from when presenting data or measuring against a SLO. Once you have the combined uncertainty, you can also calculate the expanded uncertainty to account for the level of confidence if you want. For example, if you do not have data or a spec for a vendor and do not have statistical measurement, then you need to multiplier around the confidence of what is being presented. If you know based on history of the platform over a period of time that confidence is high and variation is low, then your confidence is high. You can then multiply a coefficient (k) or multiplier times the U value to present that level of confidence based on that experience.

Navigating the challenges between on-premises and cloud optimization

There is an evolving world of complexity and paths surrounding managing performance in the cloud. As you navigate this seemingly complex topic, it can seem overwhelming. The challenge of maintaining high-performance levels while scaling complex distributed cloud applications, powered by event-driven architectures and micro-services, is no easy feat. We may often see large organizations adopting hybrid philosophies relating closer to **service-oriented architecture** (**SOA**), which breaks the benefits of using them. Using cloud technology is a journey filled with challenges and opportunities, with the constant pressure to further optimize performance, reduce latency, and ensure reliability. However, it is important not to be daunted. Understanding the intricacies of this field is not as complicated as it might initially appear though many more considerations need to be taken into account than less distributed solutions. By breaking down the various components and processes, and taking a systematic approach to analyzing and optimizing them, you can start to find your way through this maze. This book will continue to guide you, including the performance engineers and software developers, through this complex world. We will explore the realities of cloud performance, delve into the challenges you might face, and provide strategies to optimize the performance of cloud-based services. So, buckle up and prepare for an engaging deep-dive into the intricacies of cloud performance.

Moving from on-premises to cloud-based services might seem like a straightforward task, but several performance and capacity challenges differentiate these two environments and the components are different in many aspects. For one simple example, many times databases being promoted by a cloud vendor may be more limited with less functionality but lighter weight for speed. There are countless other situations where differences need to be considered around performance and scale. The information below helps you to understand these differences and prepare for your cloud migration project.

On-premises applications are discussed as follows:

- **Hardware-control**: You have absolute control over your hardware environment. This allows for better performance customization but may require substantial financial and time investment. But it gives you complete control to create a fast platform that is very responsive from tuning all of the payloads across the network, the CPU processor to be used, the control bus and buffering. You can create RAM disks or other approaches to achieving great scale with more control than operating in the cloud. You chose every aspect of the software and hardware and control everything. Much more customizable than using something like managed services in the cloud which are a black box component with little in the way of optimization. In most cases, also a very feature rich **relational database management systems** (**RDBMS**) is used. These are fast when optimized properly, but most importantly, they give h the ability to write stored procedures, which can be a godsend when it comes to creating platforms that scale well. In

good system architecture, you want to never put business logic in the database. Now that that statement is out of the way, what you can do is write **Create, Read, Update, and Delete** (**CRUD**) functions and across multiple tables.

You can make a single request across the network that can execute 50 calls in the database to aggregate or retrieve the data you need. Why this is important is because when designing high performance applications, the fastest way to create high scale is to minimize the calls across the network. **object to relational** (**OR**) frameworks have a tendency to be very chatty across the network and if each request takes 50ms and you make 100 requests, then you get to 5 seconds pretty quickly just from network traffic. This is a common junior mistake when it comes to working with large platforms. Instead of 100 requests across the wire, using an RDBMS with powerful stored procedures, you can make a single request and decrease that latency by 99 times.

- **Network latency**: Network responses are often quicker due to the proximity of resources. This is where on-premises IT solutions really shine. You can optimize the network response for payload sizes, heat up the needed connections and keep them alive and align architectural layers across the technology stack to achieve great scale and fast response times. You will notice that cloud providers do not really provide good SLO around their components with the exception of availability. This is because they were using many different types of hardware and shirt sizing component managed services that have a lot of standard deviation.

Cloud applications have been listed as follows:

- **Hardware control**: Direct control over hardware resources is minimal, leading to variable performance. This is always the challenge. You do not really have the specifications around network switches, components being used or know what is going on behind the scenes around components being swapped with virtualization or how network components operating on shared infrastructure is impacting the latency.
- **Network latency**: Performance will have a great deal of standard deviation and variation around network latency, especially in multi-region deployments. Multi-region is the largest uncertainty and we have seen latencies significantly different between going from one region to another. This is one of the most important things to look out for. If you have transactions coming from the east and going to the west region, the transactions going to be slower than transactions going from the east region to the east region. In most cases, quite a bit of latency. When this occurs, it means that configurations need to be able to handle the slower traffic backup as far as optimization and the customer experience must be understood as well as capacity. For example, if we have a reverse proxy that is routing requests from one site to another, when one site is fast and the other is slow, it means that the proxy will see more CPU time being spent on managing the slower connection and more context switching will occur across threads

and service queues need to be configured to compensate, and most importantly, if having a **disaster recovery** (**DR**) approach for traffic, you need to make sure that thread settings (throttles) in web services are set properly to handle the backup. If a failover strategy exists at the component level, then the calling component needs to be able to handle the backup. If you are counting a **Domain Name System** (**DNS**) to route traffic to a different site and that site is slower for requests being sourced in the east, then the throughput is going to be less when a disaster occurs from people accessing it from that location. This must be considered in all tuning. During a DR exercise the platform must be able to handle the end state capacity for the platform with the same customer experience. If you take a performance hit, it means impact to the customer, when it impacts the customer, it impacts usage and your company's brand.

Capacity challenges cloud versus on-premises solutions are as follows:

- **On-premises applications**:
    - **Scalability**: Enlarging capacity may be time-consuming and expensive due to the need for additional hardware. It takes time to order hardware and, in many organizations, can take up to three months to bring a server online or even more. Unless operating in a virtualized world, scaling is not as easily elastic. One benefit though is that if you take away all of the componentization and elasticity and do not use virtualization like VMware or others, and run on a bare metal implementation, performance is more than forty percent better. Hence, hardware can be much cheaper to run over time based on the hardware itself and what is required to run a piece of software Operational costs when comparing cloud to on-premises applications are higher when it comes to managing head-count for the environment in some instances, or managing short term costs around infrastructure but it depends on the organization and applications running.
    - **Use of capacity**: Resources might be underused as you need to plan for peak capacity.
- **Cloud applications**:
    - **Scalability**: Cloud services offer rapid scalability, but this could lead to unexpected costs if not monitored closely. It many times encourages poor development practices with the perception around unlimited scale and the ability to turn a knob to increase CPU and Memory. It is important to set quota limits in the cloud which restrict too much allocation to minimize the potential for cost overruns. It is also important not to let errors in software drive horizontal scaling to the upper limits. The last thing you want to do is set a limit of 200 for containers and let an error take the platform to that level. It is far too expensive. The more perception that the cloud solves all problems, because this is how the vendors have sold it, the more problems

that will occur. Organizations need to keep a tight lid on that perception and make performance and capacity planning a critical part of the infrastructure management process. This generally falls on the SRE team.

- **Use of capacity**: The cloud allows for better resource usage through the pay-as-you-go model, but it may create vulnerabilities as resources may not be readily available during peak times. When it comes to elastic scaling of applications horizontally and vertically, this can be problematic because startups need to occur around components when applications scale, which can cause some latency if not optimized properly. Where on-premises hardware and software are ready to go, cloud software is generally spun up as it is needed. To alleviate this problem, it is important to have a good scaling strategy and fast startup times for components to handle search traffic. The slower the components are, the lower the threshold needs to be around scaling limits to push components horizontal faster to account for spike traffic. Other options are to build Java or other applications natively to ensure fast startups. Let me give you an example. if you have an application that supports ordering of concert tickets and a marketing mailer goes out at 8:00am for consumers to click on a link to order tickets, then the platform will be surged with high traffic near that time. This means the platform needs to elastically scale to support the traffic.

  Let us say that the CPU utilization levels are being used to scale the cloud. If you set the threshold of utilization to 90% before spinning up another container to accept the traffic, the question becomes how long will it take to spin up that other container. Let us stay to startup a Java application takes 20 seconds. You have traffic coming in all at once and your threshold quickly reaches the 90% utilization on the first pod, then the second one begins to start. But until that second up starts up 20 seconds later, requests are still continuing to come into the first container. This means utilization is now exceeding 90% and quickly going over 100% utilization because the second container is not yet running. To accommodate this, one of two things need to happen. One is that you can lower the utilization threshold so that the next container can begin to startup sooner. The other approach one can take is to startup the Pod sooner with a natively compiled JVM or technology like Golang, RUST or C language that, because they are natively compiled from the beginning have extremely fast startup times. If you do not account for this, what happens is that you will see huge outliers in response time impacting the customer experience. Throughput will also reduce since these two metrics have an inverse relationship between one another. When doing capacity testing, using percentiles like 95th and 99th while recording the max outliers is very important. This is where you will see this and also be able to determine if the proper timeout values have been set in the infrastructure as an added benefit.

Note: Remember, every cloud migration project is unique, and there is no one-size-fits-all solution. Understanding the above differences and preparing accordingly can help smooth your transition and optimize your final performance.

Complex distributed cloud application evolution

In the modern IT landscape, one of the significant shifts we have observed is the rise of complex distributed cloud applications. This transformation constitutes an incredible breakthrough in terms of scalability, reliability, and flexibility. It also introduces new performance and functional challenges that professionals like you. Distributed cloud applications, as the name itself implies, are split across multiple computing resources. This split often will extend across various geographical locations and diverse cloud environments. It may mean multiple cloud environments interacting with one another, adding to the system complexity. Cloud applications consist of a network of integrated components working cohesively to deliver services efficiently while reducing reliance on a single centralized system. Distributed platforms on-premises were complex enough now that we can easily have multiple systems working across regions seamlessly with one another.

Let us delve deeper into what makes these applications complex. Firstly, they operate across various geographical zones, each with its unique latencies and constraints. Secondly, they are built on diverse cloud environments, which consist of public, private, or hybrid solutions, and the intricacies of security, compliance, integration, and operability all need to be considered. The need for round-the-clock availability and scalability with great performance, assuring a great user experience, pushes their complexity even further. The migration to distributed cloud applications heralds a paradigm shift from the era of monolithic applications. In monolithic architectures, all components of an application are many times tightly coupled and deployed on single or few servers. As a result, scalability and update cycles become bottlenecks. On the contrary, distributed applications, often supported by micro-services architectures, enable independent scaling and development of each service. The cloud lends itself well to distributed IT platform creation.

Are the benefits of the cloud worth the complexity? They offer scale on-demand, resilience against failures, reduced downtime, faster development cycles, and seamless user experiences across global locations. This makes the challenges worthwhile. Despite the inherent complexities, these applications have become the new norm in various business sectors, including finance, healthcare, retail, and media. By understanding the distinct aspects and components of complex distributed cloud applications, you will be better equipped to optimize their performance and tackle challenges head-on.

In the world of cloud computing, performance engineers and software developers hold pivotal roles in making it function seamlessly with great scale and performance once the

architecture supports it. It is up to them to implement code and optimize the code written with the software infrastructure in combination with SRE. You, as a performance engineer or developer, play an indispensable role in orchestrating the vast, complex machinery of cloud computing. Your role ensures that the system is not just operational but optimally functioning.

Role of performance engineers and software developers

As a performance engineer, understanding the intricate details of the cloud environment is key. Your responsibility encompasses the design, testing, and optimization stages of any cloud-based application or infrastructure. You must continuously monitor and evaluate the system's state of health, focusing on aspects such as response times, server CPU usage, memory demands, and network latency, among others. You also strive to predict potential problems and preemptively seek solutions to avoid any debilitating impact on the overall system performance.

On the other hand, as a software developer working with cloud-based solutions, your responsibilities extend beyond simple code writing. You are involved in every step of the software lifecycle, from the initial design and implementation to the subsequent testing, deployment, and maintenance stages. Your role is imperative in integrating micro-services and maintaining agility in the face of frequent updates and iterations. Your responsibility also includes working with databases, handling APIs, and ensuring that your applications are fully scalable and capable of handling varying degrees of workload. One of the most overlooked considerations by junior developers when it comes to scale is considering the scale around the database and testing with too small of datasets or understanding how collections and result sets may grow in memory. Not considering production level scale for the code being developed or testing latencies around deltas for object graphs growing in memory over time.

Despite the different focuses, both the performance engineer and the software developer, share a common goal: ensuring an optimally performing, seamless application experience in the cloud environment. It is your combined efforts that help scale applications in the cloud, manage complex architectures effectively, handle capacity challenges, and ultimately provide a seamless user experience.

As you begin to navigate through complex distributed cloud applications and EDA, a good grasp of certain fundamentals is essential. What are the most significant challenges encountered in the cloud? How do cloud applications scale? How can we perfect the art of balancing performance and functionality in the cloud? All these are among the many questions that you, as a performance engineer or software developer, need to ask and find answers to, in order to thrive in the fast-paced world of cloud computing.

When we speak about the cloud, scalability is one of the advantages often touted. The

physical limitations, is indeed a significant benefit. However, achieving efficient scalability in the cloud is not a walk in the park. It poses its own unique challenges that a performance engineer or software developer, must be prepared to tackle.

Firstly, when it comes to scaling, there is a distinct difference between vertical scaling (adding more power to an existing machine) and horizontal scaling (adding more machines). Cloud environments are particularly built for horizontal scaling, which is not without complexities. To ensure that your application runs efficiently across multiple instances, you need to consider data consistency, synchronization, communication between instances, and the overall complexity of managing many different machines.

Next, as you scale your systems, maintaining consistent performance becomes a challenge. As your system grows, so too does the amount of data and traffic; poor architecture decisions made at an early stage can come back to haunt you, as inefficiencies are magnified with scale. You need to ensure that your software is not just scalable, but also performant at scale.

The cloud offers the ability to auto-scale, adjusting resources as per demand. But setting the right scaling parameters is not simple. Auto-scaling too quickly can lead to increased costs if your demand prediction is inaccurate. Conversely, scaling too slowly can lead to performance degradation and an unhappy user-base. Scalability also means being able to handle sudden spikes in demand. This rapid scale-up can be taxing on your service and can lead to performance issues if not properly managed. It is essential to have a robust infrastructure and system architecture that can deal with these quick changes without faltering. My recommendation is that if startup is taking a considerable amount of time for a runtime environment such as a JVM, and native compilation is not an option, it is to have platforms run with min VMs/Pods set to the nominal load for the platform. Then, set CPU thresholds lower to give the next VM/Pod to startup sooner before the VM/Pod taking the traffic becomes overloaded for a surge of traffic.

While the cloud offers scalability advantages, they are not without their own set of challenges. Foreseeing and addressing these challenges from the get-go is key to ensuring seamless and economical scalability in the cloud. Let us continue exploring more concepts that can help us strengthen our hold on performance and scaling capabilities in the cloud.

Event-driven architectures

You might have come across the term EDA in your journey when learning to build scalable cloud platforms. EDA is a popular architectural pattern in the creation of highly scalable and responsive cloud applications. By diving into what event-driven architecture is and how it impacts the overall cloud performance, we can further demystify the challenges and realities of cloud performance.

At its core, an event-driven architecture is a design model where flow control is determined by events like user actions, sensor outputs, or messages from other systems. In EDA, an event triggering system, commonly known as the event

middleware component. This component then filters and routes the event to one or several event processing units, typically referred to as event consumers. The event producer is decoupled from the event consumer, which in turn increases system flexibility, scalability, and resilience.

Why should architects and software developers embrace EDA in their work? EDA can improve cloud application performance in many ways. The following are some key take aways:

- **Asynchronous communication and improved user experience**: Since the EDA model operates on a non-blocking, asynchronous communication method, it does not impact the user experience. Users do not have to wait for tasks to be completed before they can proceed, resulting in a smooth and seamless experience.
- **Improved system responsiveness**: EDA promotes system decoupling. This leads to multiple, independent elements that can effectively respond to each event, which in turn increases system responsiveness.
- **Elevated scalability**: With EDA, it is easier to scale systems vertically and horizontally.

Micro-services

Micro-services architecture has emerged as a key solution to scaling distributed cloud applications effectively. This architecture strategy breaks down complex applications into smaller, more manageable pieces—each one a functionally complete micro-service that can be developed and deployed independently.

Unlike traditional monolithic applications, where the components are tightly coupled and operate as a unified system, micro-services are designed to operate independently. Each micro-service runs in its own process and communicates with others via HTTP/REST or asynchronous messaging. This decoupled structure is advantageous, as it enables independent scaling of each service based on demand, thereby optimizing resource utilization.

The benefits of micro-services in the cloud are as follows:

- **Scalability**: Micro-services can easily be scaled up or down independently depending upon demand. This way, if a particular service requires more resources, only that specific micro-service needs scaling rather than the entire application.
- **Flexibility**: Each micro-service can use a different technology stack that best suits its functionality. This allows developers to employ the best tools for each service.
- **Fault isolation**: In a micro-services architecture, if a service fails, it does not bring the whole application down. Only the functionality provided by the specific failed service will be impacted, making the application resilient and easily recoverable.

Adopting micro-services for cloud applications

While micro-services architecture holds great promise in increasing scalability and performance of cloud applications, adopting it is not without its challenges. Developers should consider several factors, including service coordination, data consistency, and network latency. Creating an efficient communication system between micro-services is essential, as they need to share data and handle requests. Ensuring consistent data across independent services can be tricky too, this is usually handled through some form of data replication or synchronization, which can be complex to manage. Lastly, as services may be spread out over a cloud network, network latency can impact performance and should be taken into consideration while designing the architecture.

Moving to a micro-services architecture is fundamentally about striking a balance between the potential benefits and the challenges of its adoption. By carefully considering these aspects during your planning process, you can leverage the power of micro-services to effectively scale complex cloud applications, tapping into unprecedented flexibility and fault isolation capabilities.

Undoubtedly, one of the most pivotal balance beams that performance engineers and software developers have to traverse in the cloud environment is between performance and functionality referred to as features. But remember that customers in many cases rather have speed and simplicity and something polished with resiliency over something that is overly complex with too many features. The core features are what is important and are the primary revenue drivers over time. Nonfunctional and functional requirements need to be considered in every product architecture. Quality engineering also comes into play with features being performance tested to requirements and either working with performance engineers or developing their own automation around performance measurement. Every decision made in designing and deploying applications can tip the balance, affecting the user experience of your cloud applications.

When you develop and deploy applications in the cloud environment, you meet the constant demand of delivering more features and seamless user experiences. At the same time, maintaining high performance becomes non-negotiable. Unfortunately, these two elements functionality and performance tend to resist each other challenging the non-negotiable premise. More features often mean more lines of code to execute, which could affect overall application performance. On the other hand, an extremely optimized application may overly limit what an IT platform can do, thereby compromising functionality. Hence, striking a balance between these two important aspects becomes vital.

The term sometimes used in the industry is the conception of a performance-functionality trade off. This is a fitting description for this scenario. It refers to the process of weighing the cost of added functionality against the potential performance hit. For example, a feature might make an application more appealing or user-friendly but can also make it slower, reducing effectiveness or productivity. There needs to be a balance.

Walking the tightrope

One analogy that we have heard in the past is that navigating the delicate equilibrium between performance and functionality is similar to a tightrope walker carefully placing each step to avoid a catastrophic fall. In my opinion, this is a great analogy and one that you should consider when understanding resiliency of IT platforms. Much like that performer, you must find the right balance and continuously adjust to changing conditions. Just like when we talk about continuous performance measurement or continuous functional testing, we must consider continuous resiliency testing and understand the breaking points or rephrase degradation points in the platform. When the platform degrades due to perhaps a failure and needing to route traffic to a different network path, this impacts latency which impacts throughput or the processing rate for the platform. This change in performance dynamic needs to be accounted for by optimizing the platform to support these changes. For example, if all traffic routes from one region to another region and latency significantly increases for that traffic across the network, then the source must be able to handle the backup in threads, and the destination region must be able to support the additional processing threads without significant queuing of requests. This optimization is one of the most valuable aspects of implementing chaos engineering in an organization.

In this, understanding your application's performance profile and capacity limits is as essential as balancing on a tightrope. On one side of the tightrope, you have the aspiration to deliver compelling features and superior user experiences. These are the elements that make your application valuable and impactful. On the other side of the rope, you have the need for optimal performance, efficient use of resources, and swift response times. These elements ensure user satisfaction and service reliability. Moving too far in one direction can lead to application bloating, where excessive features slow down response times and push the system's capacity to its limits. Leaning too far in the other direction risks oversized infrastructure with a lack of features that may lead to a thin, unengaging user experience.

To walk this tightrope, you should adopt a prudent and methodical approach to application design and scaling. One strategy is to prioritize features based on their value to users and their impact on system performance. By thoroughly benchmarking and monitoring your application's performance continuously, you can make informed decisions on where to allocate resources and how to scale effectively.

Note: Remember, the balance between functionality and performance is not a one-and-done achievement. It requires continual adjustment, routine testing, and frequent re-evaluation of priorities and resources. Much like the attentive steps of a tightrope walker, you too must constantly assess and adjust your steps to efficiently tread the thin line between performance and functionality.

Chaos engineering

As you soar through cloud computing's expansive landscape, resilience emerges as a cornerstone of robust performance, even in the shadow of inevitable failures. Failure is a reality that accompanies all technological endeavors, including your complex distributed cloud applications. Precisely managing these failures, rather than merely avoiding them, underpins a resilient system. High performance, then, can be ensured even when in the face of adversity. As a member of the rapidly evolving tech industry, you must be intrigued by something referred to as chaos engineering, a burgeoning concept that is reshaping the digital landscape. We know that you have likely heard the term buzzing around your IT department, but are yet to delve into why it matters and how it is used in practice.

Chaos engineering is the science of performing intentional experiments on a system in order to learn its weaknesses and build its resilience against unpredictable conditions. It is a preemptive, systematic approach that puts your infrastructure through the wringer to ensure it can withstand irregularities without a hitch. The ownership of chaos engineering and the strategy is by SREs who play a pivotal role in its execution. Their mission is to optimize infrastructure and minimize degradation. Remember, the goal is not to wreak havoc but to fortify the system against turmoil and understand how to optimize the platform to operate in a degraded state without impacting SLO. Resilience requirements set the benchmarks for the ability of your system to recover and maintain functionality after experiencing stress or failure. Building this capability insulates your customers from what is going on internally on your IT platform. Remember, what is really important is what the customer sees from a performance and user experience perspective.

The design and implementation of resilient systems hinge on several principles:

- **Fault tolerance**: Software and performance engineers aim to construct a system that, despite the onset of a fault or failure, continues its core function with good performance while taking steps toward recovery.
- **Redundancy**: This involves the duplication of critical components or functions of a system with the intention of increasing reliability. In the context of the cloud, it could involve multiple distributed components working in tandem. When traffic goes from a primary location to a redundant component that extra network hop or degradation if going across regions needs to be considered. Through keeping components active and connections active along with the proper thread configurations the failover should be seamless and still operate within the SLO for the platform. The redundant system needs to be able to scale to be able to handle all of the load if required. Hence, when tuning an IT platform that is balanced across two sets of the same infrastructure or across regions, we consider that both sides need to be tuned to be able to handle the full workload and be responsive. The customer should not be able to see what is happening being the scenes.

- **Failover systems**: These systems are standby operational modes which automatically switch on to perform functions in case of system failure. Sometimes components like databases may not be active in both sites or both be active, but not fully in sync with one another. This means the redundant system needs to be brought online and data or other components brought up to be active. This is critical for disaster recovery. In this case, you want redundancy in different data centers or across a long distance in different regions. You have probably heard the term DR if you have worked in IT. It is an important concept, especially when working with mission critical business applications. DR refers to the strategies and procedures that ensure the rapid recovery of an organization's technological infrastructure and systems following any form of disruption. In other words, if a disaster disrupts a company's normal operations, DR steps in to help that company get back on its feet as quickly as possible. This is typically achieved through the use of backup systems, fail-safe measures, and meticulous planning. So, whether it is a major power outage, a cyber-attack, or a system failure, disaster recovery aims to minimize the impact and ensure that the organization's critical operations return to normal in a short amount of time.

Understanding the tenets of chaos engineering is a game-changer. Chaos engineering, sometimes referred to as the discipline of experiments on a system to build confidence in the system's capability, plays a vital role in ensuring cloud performance and resilience.

Chaos engineering is thinking like an attacker and proactively injecting harm into the system to assess if the system can sustain, survive and recover. It is the art of injecting controlled chaos into an IT platform to identify different points of failure before they escalate into real-world problems.

Firstly, you begin by forming a hypothesis about how your system should behave under disruptive conditions. It is important to understand the architecture and expected performance of the system and combine chaos experiments with chaos engineering, thus establishing a baseline for normal operations and injecting Chaos scenarios under load to understand behavior. You design and execute controlled experiments, typically starting with non-threatening scenarios such as shutting down a single Kubernetes pod and observing the system's ability to absorb the shock. For example, were the other pods able to absorb the load and the system still meeting the service level objectives. Did any transactions fail while the platform was under load and the pod went down? What happens to inflight transactions? Were data consistencies impacted?

After executing defined experiments, it is important to monitor and analyze the system's behavior using observability tools. This process unveils insights into your system's vulnerabilities, performance bottlenecks, and unexpected interdependencies. Analyzing these outcomes guides you to make informed decisions on strengthening your infrastructure, such as improving load balancing or implementing more robust fallback mechanisms. It also stresses tests error handling in the application, retries, logging, and many other non-happy path code runtime scenarios.

Remember that most platforms are not running in a vacuum. Thus, chaos engineering thrives on collaboration with dependent teams. Involving developers, SRE teams, and business analysts ensures that you gather diverse insights and solutions when unexpected issues surface. In any organization looking to develop robust products, adopting a culture of continuous improvement is essential. Iterating on chaos experiments based on gathered insights ensures that your systems become progressively resilient against unforeseen disruptions. In a nutshell, minimal, simplistic chaos engineering involves:

- **Culling the herd**: This proactive approach means systematically and purposefully shutting down instances in your architecture to make sure your system skills up or down correctly in response.
- **Seeing the system under duress**: Chaos engineering allows you to see what happens under conditions of stress and to ensure the fail-over systems work as designed.
- **Learning from failure**: Each chaos test presents an opportunity to refine your approach and gain valuable insights into the resilience of your application.

Chaos in action

How exactly does chaos engineering play a part in ensuring the robust performance of distributed cloud applications? Chaos engineering, in its essence, is an innovative approach to testing distributed systems by introducing turbulence and, observing how the system reacts and then optimizing infrastructure and software to support a level of degradation or offsetting that degradation for the customer. Generally, SLO determines the degradation level. For example, as a general rule of thumb if the platform degrades by twenty percent, are the SLOs around performance and resiliency still met? The inherent distribution and decentralization of cloud applications present an advantage when it comes to building resilience. Here, resilience can be afforded by designing microservices that isolate failures and prevent them from cascading across the system. Furthermore, you can leverage the cloud's scalability to quickly swap a malfunctioning service with a working instance.

Once an effort has been made to establish a resilient system, that resilience should be tested. Chaos engineering has emerged as a powerful practice here. By deliberately injecting faults into a system in production, engineers and developers can observe how the system reacts, providing valuable insight into its resilience. Building resilience is not a one-off task, but a process of continuous improvement. With every failure, your system should evolve, using the feedback from failure events to build a stronger, more resilient system capable of upholding its performance.

Chaos engineering practices should be continuous and automated and discoveries will continuously be made as the platform evolves. Using practices such as this is how to build robust high performing platforms. You learn from the experiments and augment the code to account for what you have learned to allow it to be able to handle the degradation

gracefully and never impact the customer. For a well-designed platform, it should be impossible to bring down the platform. The platform should be designed with enough rigor to handle failed components or an attack of being overloaded with requests. This does not mean scale the platform to support a hundred virtual machines, it means throttling or restricting requests coming into the platform, it means setting timeouts at every layer and taking smart measures to prevent the backend platform from being overloaded. If you have a micro-service that is written in Java, it should not be possible to send it enough traffic to overload it and bring it to a crash state.

Chaos engineering helps expose many limitations around platforms. The practice started when *Netflix* started executing it with the tool *Chaos Monkey*, but these days there are hundreds of easy-to-use tools in the cloud and on-premise to inject the needed degradation experiments to support continual resiliency testing. You can build it into your build pipeline along with performance and functional quality gating. You do not need to go out and start using an expensive third-party solution. There is a great deal of free open-source solutions for working in the cloud.

Conclusion

In conclusion, you know understand the benefits of developing new IT platforms in the cloud and when on-premises solutions should be considered. You understand the benefits of event driven architecture and micro-services with respect to performance. Most importantly you learned the importance of resiliency testing and chaos engineering when building high performance platforms. This has just been the tip of the iceberg. Now, we are going to go even deeper and improve your understanding around optimization in the cloud.

In the next chapter, we move into cloud scalability and how to optimize platforms. We will talk about the techniques used to optimize performance and throughput while minimizing cost. We will also re-iterate the importance of tuning to support full failover of load during a disaster or high availability situation across components. Capacity planning for IT platforms will be discussed in detail and how optimization can further drive more capacity and a build better user experience. Let us get started!

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 4
Cloud Environment and Scalability

Introduction

This chapter of the book will introduce you to scalability and capacity planning. We will discuss some of the challenges and realities around managing cloud capacity and the processes to drive high level scale at low runtime cost. We will discuss scaling Kubernetes and the different scaling types. We will talk about the practice of capacity planning. Remember that this chapter continues to build on previous chapters to round out your understanding of performance engineering. Though you are being introduced to a great deal of concepts quickly, it is important to understand these prior to reading the next chapters which go even deeper into these topics.

Structure

This chapter covers the following the following topics:

- Capacity planning concepts
- Network tuning
- Playbook for performance best practices
- Scaling cloud architectures

Objectives

In this chapter, we delve into important concepts in computing; Little's law, Amdahl's law, the **Universal Scalability Law** (**USL**), and the art of capacity planning in the cloud. We hope this chapter provides you insightful knowledge into these core theories which underpin efficient cloud computing, and guides you in effective capacity planning. Understanding the laws around capacity planning and cloud computing is extremely practical. It helps with making strategic decisions and optimization of resources for the highest throughput. Let us not wait any longer and dive right into it!

Capacity planning concepts

Let us start with a brief overview:

- **Little's law**: Focuses on the relationship between the number of tasks in a system, the system's throughput, and the time to complete each task.
- **Amdahl's law**: Highlights the limitations and benefits of parallel processing and the importance of task synergy in a system.

Capacity planning involves balancing costs through efficiencies to achieve the maximum scalability. It is not just about allocating maximum resources to achieve scale. Instead, it means making more informed decisions to align your cloud resources with your business requirements to keep costs as low as possible while achieving the best possible experience for the customer. It also means being able to handle the user behavior throughout the year through holidays and special events or processing. You want to shoot for understanding the peak capacity needed over a twelve-month period at a minimum. To achieve this, it requires doing regular capacity testing to understand blockers to capacity and optimization for your platform ongoing to keep costs low and to keep information available for at least twelve months of time. Once you understand capacity planning, you will be positioned to make informed decisions for your cloud computing needs. One of the fallacies commonly seen in planning for capacity is the point at which capacity is measured. Sometimes organizations feel that at the point of failing hardware or virtual machines is the point of max capacity; so, when requests going into the IT platform stop receiving responses back. It is not, we want to be clear about this. For any platform, the capacity should be considered the point at which the software is no longer meeting the **service level objectives** (**SLO**). This is the point where the platform is considered no longer usable by the customer.

Understanding capacity planning for cloud computing would be incomplete without mentioning two vital laws of performance computing: Little's law and Amdahl's law. Both mathematical models are used in cloud capacity planning to help optimize resources for better efficiency. Remember, cloud capacity planning is mindful anticipation. The application of Little's law and Amdahl's law in this context allows you to model, predict and scale based on your specific workload and resource utilization, helping optimize your cloud strategy.

First, let us explore Little's law. This is one of those mathematical equations that aids you in predicting the performance of your system in the cloud. Based on arrival rates and waiting times, Little's law helps to pinpoint the average number of customers in the system (L), which is equal to the arrival rate (λ) times the time a customer spends in the system (W).

The equation reads as:

$$L = \lambda W$$

Knowing these relationships can be instrumental in capacity planning as it assists in understanding what to expect from your cloud resources. Many analytical tools can be instrumental in gathering the needed metrics to understand platform capacity. *Google Analytics*, for example, will help you understand how many customer sessions are active and how log users are staying on the platform. Many **real user metrics** (**RUM**) vendor solutions provide the same benefits. This information used with the formula for Little's law can help you determine the capacity for the platform. You will be able to understand how fast users are coming into the platform and how fast they are processing requests through the various user journeys.

Next, we have Amdahl's law, named after *Gene Amdahl*. This law considers the aspect of parallel processing in computing. Simply put, this law states that the improvement to be gained from enhancing a particular aspect of a system is limited by the fraction of the time that the improved aspect is actually used. You focus on the critical bottleneck and this factor becomes apparent with regard to system performance and efficiency, presenting considerations for effectively scaling up a cloud system. For instance, if a component of your system that represents 50% of your workload is optimized to be twice as efficient, the maximum overall system performance improvement can be significant. That means you did not optimize everything; you found the key bottleneck to scale and optimized that component or dependency on user journeys used in the application. Amdahl's law states that the overall speedup of a program, when a certain fraction of it is made faster, is given by:

$$Speedup = 1 / ((1 - P) + (P / S))$$

- P is the fraction of the program that is made faster
- S is the speedup factor for that fraction

Capacity planning involves much more than these two laws, you need to understand how to size your organizations allocated servers or computing resources to support the businesses incoming traffic pipeline. For example, if you know that you have fifty thousand more customers coming onto the platform, you need to understand how this impacts server utilization. The primary server provisioning strategy for most organizations, before the cloud or virtualization, was to over provision by more than you would ever need to be safe. The minimum increment you could increase capacity by, was typically one full server, which had some repercussions. This automatically meant deploying more resources than an organization would need ahead of time. If, for some reason, the capacity

analysis was wrong or the input to the capacity model was wrong, you would end up frantically optimizing software within the existing defined footprint. Undoubtedly, you would end up pushing for more servers vs. less as a default server provisioning strategy.

The cloud has simplified the provisioning of servers, which means that we need to continually optimize and allocate exactly what we need to keep costs in check. Tuning servers and software are an ongoing effort and an important one. Remember to consider all scenarios for hardware and software when it comes to capacity planning is based on planned or historical observations.

Why optimization is important

When it comes to optimization of servers and software, many organizations do not consider tuning their servers beyond happy path scenarios, which causes problems. Instead, they order more hardware with a standard sizing and then use software to monitor the servers as they scale horizontally across balancers and plan on adding more and more servers horizontally as needed. They invoke models in monitoring software, add Data Scientists to staff and define alert thresholds in their modeling software. When failures occur on the platform, hardware is thrown at the problem, which is costly over time. Generally, in the short term, this costly approach many times solves the capacity problem. The problem is that the software inefficiencies not resolved would snowball exponentially beyond what some proper analysis could have achieved early on.

When you think about optimization in an organization, never think about being reactionary. Think in terms of good IT processes, ongoing optimization and refactoring as products are developed. Budgeting five or ten percent bandwidth for refactoring and optimization can solve a lot or problems early in the development cycle. The problem with throwing hardware at an issue is that the inefficiencies in the software tend to get much worse over time. But how resilient should a platform really be anyway?

Pipeline to traffic pattern correlation

When evaluating your capacity planning, understanding a few concepts from the very beginning is important; the first concept is to understand inbound traffic patterns across each tier of the distributed platform concurrently at peak levels in production. The next is to understand what the traffic pattern will be in the future, and finally it is important to understand the potential bottlenecks that currently exist in the platform, capacity drivers. For distributed platform capacity, it all comes down to queuing patterns and interconnected endpoints along with coherence/loss associated with this. Modeling the software and understanding how traffic patterns, queue depths, concurrency and latencies correlate is very important. In most cases, this model is not linear and is dependent on coherence across multiple processors, balancing, network, and data patterns.

Part of those patterns is understanding negative scenarios. Most companies will generally

expected happy path traffic pattern scenarios and base that scale on testing in lower environments that, in many cases, may not be aligned with the production environment. This is unfortunate and is typically a situation that drives capacity overload situations because the true capacity is being measured incorrectly.

Negative scenarios are just as important as positive scenarios and is the real world we all live in when it comes to IT platforms. No platform can expect perfect user behavior or one hundred percent reliability. Nothing in today's world will be a full happy path operation regardless of how many life coaches or high paid consultants we choose to embrace to try to make that the case. You do not drive down the highway and never run into traffic or an unexpected event that may disrupt your path. You do not get your morning coffee without standing in a queue waiting for my turn to execute an order for your beverage; unless it is bad coffee, but that is another discussion.

Patterns, for example, like logins. If you have a platform that accepts logins from users and a request comes in with a user id and password and sixty percent of the time the customer enters the information in an invalid way and has to reenter it, it means that this negative flow along with any associated exceptions or logging needs to be performance tested and the impacts to capacity fully understood. This is a very common one that is often overlooked by organizations. Most platforms have a very high level of failures in logins due to customer behavior, but in many cases, teams will only test with successful logins. This is a big mistake and is only capturing forty percent of the workload during a performance test in this example. Understanding negative traffic patterns is the most often overlooked failure in proper workload modeling for capacity tests. It is also what drives the most uncertainty around understanding a platforms capacity. Having a proper workload model that fully represents the percentage of expected negative scenarios in each user journey is the only way to have valid results.

Testing environment similar to production is important

The other item that can impact the validity of testing that is common is testing in an environment that is different from production or is not a valid subset of the infrastructure to be able to extrapolate from. Let us say you have ten virtual servers running in production, but your organization wants to keep costs low, so only two servers are available in the lower environment to test against. In this situation, you need to be able to reliably test with the two servers and mathematically extrapolate the result to represent five times load. If you achieve two **transactions per second** (**TPS**) in performance testing as a processing rate for example, you need to know if this truly translates to ten TPS in production. You may find that filter or rewrite rules are different in the balancer in one environment, perhaps the balancer even has a slower CPU. In this case perhaps two TPS in the lower environment translates to eight TPS in production.

Remember that extrapolation is all about math and measurement. Where you have vendor specs, you can take into account loss mathematically and

it means you need to profile and measure to understand the differences. Differences in compute power of the processor being used, the bus frequency, L1/L2 cache latency, memory access times, disk access times etcetera between lower and upper environments must be understood. You also need to understand the loss you achieve across balancing across nodes and the differences around bandwidth and network latency as well.

Balancers, reverse proxies, network switches, firewall rules all come into play when it comes to latencies. You need to be able to traceroute or use utilities like MTR, which is a Linux utility, to average ping latencies across network hops over time to profile the different network paths and understand the latency dynamics. Profiling these latencies and doing focused testing to understand the dynamics is important to understand for calculating extrapolation around true scale for production when using a lower environment that is different. One thing you do not want to do is test in a lower environment with different server configurations, JVM configurations, different database sizing, etcetera. That type of test is a waste of time and will have everyone chasing down red herrings. If the test results are good, they may not really be and you have a false sense of security. If they are bad, they may be bad due to the deltas in the environmental differences with production. Instead, you need to first understand the differences and scale down the test, set think times, latency variation etcetera accordingly. You need to know that when you test send requests at five requests per second, which may equate to three requests per second throughput rate and 500 millisecond response times, what these values translate to in production. If you are using simulators in the lower environment these should represent a Galton distribution pattern that represents the load in production so that the test will be accurate. Using tools like WireMock, which is one of the few mocking tools that does a good job at performance, is something to consider when this is needed.

Understanding percentiles

You also need to understand percentiles and response time curve differences between the two platforms. If you say that you can achieve ten TPS as a processing rate with 500 millisecond response times in the lower environment you need to understand if those five hundred milliseconds are average, 50th, 75th, 95th, or 99th when it comes to metrics measurement. The best approach is to measure each percentile and extrapolate the differences. But at a minimum, you want to understand the 95th and 99th percentiles and max outliers around response times for a given processing rate. Understanding the metrics between 95th to the maximum outlier will help you understand what users experience worst-case. The max outlier will also tell you the proper timeouts have been set for the platform. If you see an outlier that is five minutes long, this means timeouts have not been set and need to be. Timeouts ensure that you will meet your capacity requirements and not overload the platform, impacting everyone. You should not see a max outlier above the timeout setting. Percentile metrics help you fully understand the worst-case experience for five percent of your users with an understanding that all metrics below that percentile are less. If you measure a response time for a web page to load of five seconds at the 95th percentile, it means that 95% of all

5 seconds. But do not minimize this when it comes to importance. If five out of every hundred users having a bad experience, that should never be minimized. It means that out of 10,000 users, 500 are having an awful experience. Sometimes organizations will even do performance measurements and set SLO at the 90th percentile. This means that the objective is to have a response time of 2 seconds and that ten percent of your customers are having a worse experience than what is measured against performance quality gating. You need to understand that user experience and why it is happening.

In most cases, you will find it is that timeouts are not set, containers or JVMs are not starting up fast enough for the spike of traffic coming in or another anomaly that you should be aware of. You need to always understand why. When it comes to 90th percentile and things being slow beyond that. If you have ten thousand users on a platform, you do not want a thousand of them hating your product and brand. You want to optimize for the 95th percentile and eliminate the outliers that are occurring above that percentile as best you can. Beyond the 95th percentile is generally where something in the platform is causing outliers due to not being optimized, such as JVMs starting too slowly or timeouts not being in place. This percentile level is a good level to use.

Outliers are what tell you where you misconfigured your infrastructure or did not do something efficiently in many cases and many times it is low hanging fruit. It may be a bottlenecked thread downstream or slow startup time for containers or JVMs. You want to startup containers sooner and timeout any bottlenecks and give a good message to the customer. No customer wants to wait because a screen is hanging. Timeout the transaction and just say the platform is experiencing high volume, please try again later or handle it in a graceful way or have a process that executes something later, though that can come with its own set of challenges in a real-time platform. You want to give the impression that it is a popular product and user-friendly product when it comes to all error messaging. But remember that optimizing to 95th percentile overall and then minimizing the outliers is what you need to achieve to keep a great user experience, keep costs low, and be able to understand capacity.

Design for interruption and short-term events

What happens when an unexpected event occurs in your life that involves waiting or changes in traffic patterns? Do you fall over? Does your vehicle stop operating in traffic when things slow down or when you have to swerve to prevent running into that squirrel that ran in front of you? When it comes to servers, without the proper resiliency planning, unexpected events will undoubtedly occur. Unexpected short-term events occur regularly, and being able to handle them well attests to someone who has done the proper planning for an organization. It also means being able to easily absorb disruptions in service or long latencies in service due to downstream tiers encountering high volume or intermittent failure conditions. These maybe servers that you do not have control over. Servers that you manage should be scaled to expect a certain number of failures and backups while not falling over due to a short-term interruption. In reality, each tier of a distributed platform

should be properly tuned both from a software and hardware perspective to handle the worst-case event expected for the platform. We will talk about how to achieve this later.

For inbound traffic into an organization network or in the cloud, one of the most common oversights we have seen is that organizations do not throttle balancers and set the needed connection, thread, queuing limits, timeouts, etcetera (F5s, reverse proxies, etcetera) and restrict the incoming load into their environment using pools or concurrency limits. When an organization does not have any restrictions, this traffic can come in and bring down servers to their knees. Many times, operational groups will push the throttling/concurrency request management responsibilities downstream to an application team but in reality, this is a terrible idea, because it hogs the network resources internally to an organization, impacting other areas unrelated to an application. Beyond proper inbound throttling, software should be developed with resiliency in mind, the servers themselves need to be scaled with the right failover, caching or queuing abilities to be able to handle a disruption should a backup occur. It is this need for resiliency in both servers and software in combination that has made software like Chaos Monkey so popular these days.

The servers themselves may be in a local data center or distributed across multiple cloud environments or a combination. Servers these days are, in most cases, running optimized containers like Docker or self-contained TCP/IP services like Spring Boot or Golang with embedded socket listeners, waiting for an inbound request or process. The challenge is many times when it comes to optimization is thinking in terms of micro-services and smaller throttled instances using lower memory footprints versus larger, more traditional service containers.

Generally, container cluster management software like *Kubernetes* and *Apache Swarm* will need to be configured to optimize resources on a node while container instances themselves are tuned with much smaller memory foot prints. Then, they are throttled accordingly to support the inbound load. You generally do not think in terms of creating a 10 gigabyte heap for a Java container in the cloud these days for most applications; instead, you think in term of creating a 1 gigabyte instance, throttling it to those thread resource limits/queue depths, etcetera and having ten copies balanced for the service that you ramp up and down based on traffic patterns across as many Clouds as you need to. Remember though that cloud management software performance matters. Container runtimes combined with Virtual Machines run slower than containers over native (about 40% slower).

These days organizations are pushing to the cloud faster than anyone ever thought possible. The new cloud paradigm lets DevOps teams deploy new resources quickly when needed. With proper containerization they can also do it in a very granular fashion incrementally adding small amounts of just the right number of resources (if containers sized, designed properly). Running platforms in the cloud can be a scary place for planners when it comes to capacity considerations because it can be easy for developers to not consider the high costs associated with waste-fullness and inefficient coding.

Scaling across distributed platforms

The temptation is strong to fight any capacity problem by simply throwing more money at it. That is often one strategy, but the other requires us to get better at doing capacity planning for the cloud and reduce costs. Engineering platforms for great performance and capacity will always be required and have become a mainstay due to the expectations of young generations around application latency. For IT platform capacity, we need the same due diligence to keep costs at bay and ensure the proper efficiencies.

Most large distributed platforms end up leveraging an n-tier architecture. These architectures typically have a large number of application servers talking to a large number of NoSQL horizontal nodes or a relatively small pool of large vertically sized relational databases. This is overwhelmingly the most common model because it allows you to take advantage of horizontal scalability on your application server layer for most processing. Then it requires only vertically scaling of the database if using relational. Though many relational databases now support sharding going with a similar model to NoSQL and going horizontal as well through a balancing/routing mechanism (Postgres, etcetera). Generally, though, from what we have witnessed, NoSQL products like MongoDB, etcetera is the way to go for real-time applications catered towards a specific purpose. Applications using a service-oriented or micro-services architectures generally find a communications decoupling mechanism makes it easier to resize services at will or add new services as the application's needs change.

These mechanisms control/orchestrate processing requests at different rates. For the application layer people used to leverage MQSeries, SonicMQ and open queuing mechanisms, but these days to scale messaging most folks leverage tools like NSQ or Kafka which allow processing distribution across multiple servers. These frameworks are very robust. When working in the cloud or designing distributed platforms, decoupling and proper separation of concerns are the most important design consideration around containers. Containers need to run independently of one another and not be aware or integrated in a fashion to be directly dependent on a particular instance. How do we do capacity planning for something so complex?

Capacity planners always leverage mathematics to determine hardware scalability and model software components and subsystems regardless of configuration to determine how they scale/extrapolate to other environments. They also commonly leverage monitoring tools that look for outliers and Gaussian models/AI to springboard events when capacity starts become worrisome due to unforeseen events. Due to the complexity of transaction distribution, many times, people struggle to determine cloud capacity or understand what-if types of scenarios around it. Then they start doing performance engineering and try to correlate what performance results mean with respect to capacity or even in some cases have difficulty understanding the differences between performance engineering and capacity planning activities.

The challenge comes for many people as a result of the complexity around distributed traffic, which may bounce off many endpoints and across tiers and be partially processed on different processors with different instruction frequencies, cache and bus rates. Many organizations will look for simple tools that tell them where they are at as far as thresholds, but become challenged when it comes to predicting what load levels will exist when it comes to software. In distributed platforms, each tier processes transactions at a different rate and if the rate is different enough, queuing patterns are used to buffer or act as a **first-in-first-out** (**FIFO**) queuing pattern to put in the necessary wait to distribute downstream actions or distribute work asynchronously across many systems to parallelize activities to help achieve the needed throughput.

Distributed platforms tend to process requests in different service tiers that reside on different physical or virtual servers; in most cases with their own independent queuing and thread pool configurations. Each one of these TCP/IP requests coming across the network generally utilizes one or more processing threads in each tier to handle the work, and those processing threads each impact CPU time.

CPU Time is the time the processor on a computer spends running instructions on a given **process identifier** (**PID**). If you learn nothing else, remember CPU time. This is the real key to understanding capacity, modeling and performance. That simple time you see when you do a **ps** command in Linux against an individual process running during a peak timeframe becomes the holy grail at understanding the world of performance engineering and capacity planning. Hardware scalability overall on platform with multiple processors can always be computed using the USL[1] and Amdahl's serial fraction theory[2].

The USL is a mathematical model developed by *Dr. Neil Gunther* to analyze and predict the scalability of computer systems. If you want to be a great performance engineer or understand capacity well, *Mr. Gunther* has some great books on the market to talk about platforms and queuing patterns. USL provides a framework for understanding how the performance of a system changes as the workload and resources are scaled up or down. The USL equation takes into account the major dependencies to understanding an IT platform's scale: Throughput, concurrency, and the degree of serialization of tasks in the platform. Throughput refers to the number of requests or transactions that a system can handle in a given time period. Concurrency represents the level of parallelism or the number of concurrent users or processes. The degree of serialization measures the extent to which tasks in the system are dependent on each other. The USL equation is expressed as: $S(N) = N / (1 + \alpha(N-1) + \beta N(N-1))$, where $S(N)$ is the scalability of the system with N resources, and α and β are coefficients that capture the impact of contention and serialization. By fitting this equation to observational data, it is possible to estimate the values of α and β, and thus predict the scalability of the system under different conditions. If you want to go deeper, we would encourage you to read up on this formula and understand the dynamics around universal scaling.

1. http://www.perfdynamics.com/Manifesto/USLscalability.html#tth_sEc1

But what is important here is not just understanding these performance/capacity laws from a hardware perspective, but from a software perspective. Understanding both the hardware and software the service time and queuing dynamics relationships with respect to multiple processor/balancing coherence is important. This, along with understanding how thread pools impact queuing times, generally allows you to accurately project capacity for all platforms in the industry. Why is it important to understand CPU time and *Gunther's* USL/Amdahl's serial fraction theory?

Understanding the relationships to computer resources is the key much more than the formulas. Now, to explain this, we hear the term over and over again, scalability. What is scalability? What does it mean? A lot of people use the term scalability without clearly defining it, let alone defining it quantitatively. Generally, platforms stop scaling when response times start increasing under load as they start to bottleneck due to contention against the same resources. But computer system scalability must be always quantified in some way. In reality, if you cannot quantify it, you really cannot guarantee it. Every queueing facility in a performance model requires understanding of service time. This is why CPU time and inherently queuing times are so important and why we mentioned the importance of CPU Time earlier. Without the appropriate queues in the model and understanding service and queue times at each tier of a distributed platform, response times and scale cannot be predicted. Beyond the more granular relationships of the standard formulas around capacity that can sometimes feel ambiguous to some people, you can also generally ballpark capacity projections pretty accurately using the formula below and add in a loss factor for balancing. Let us assume a worst-case loss factor of .8 for example balancing across 4 servers just as an example (measure it to determine the actual loss factor).

Transactions Per Hour Capacity = (3600 x number of processors x %CPU Utilization for a PID/ ({PID CPU Service Time + PID CPU Run Queue Time)) x Balancing Loss Coefficient (.8)

You can even take this further and add in SPECrate from the standard organization for hardware performance (**https://www.spec.org**) and augment the formula as follows:

*Transactions per hour capacity = ((3600 x number of processors x %CPU utilization for a PID/ ({PID CPU service time + PID CPU run queue time)) x balancing loss coefficient (.8)) * (SPECrate of new processor/SPECrate of current processor)*

Software capacity

Hardware capacity calculations are very easy compared to software capacity. Software capacity is the highest rate of processing where you still have a usable system. Yes, we said usable! This is really what determines capacity; focusing on TPS requirements and acceptable/usable response times. When response times go up, TPS goes down and that is generally how you determine where your software is no longer scaling. If transaction queuing starts occurring during backup events, web thread suspensions start occurring, context switching occurs much more often, and eventually, the CPU on servers starts hitting

much higher utilization. In the meantime, the customer experiences deadlock. A 20-second latency downstream can easily cause a completely failed transaction or multiple-minute wait due to thread suspensions.

We are all familiar with *Gunther* and *Amdahl* and the queuing patterns associated with managing capacity and throughput. Many times, the art of capacity planning spills over into performance engineering, and it is very important to understand how to size servers in a distributed platform. Most importantly, in order to plan for capacity, it is important to understand the upcoming pipeline for new loads and look at the past history of backup events to size servers accordingly. The primary goal for any organization or distributed platform is to be able to readily absorb the load when downstream server's backup up to the timeout of the downstream server without exceeding capacity of your upstream servers. You can generally determine how to do this using past behavioral patterns of your platform. Many times, the downstream servers are owned by other organizations or entities that you have no control over, which makes it difficult to size your own servers. This means understanding worst-case arrival rates and processing times over the past 12 months and planning/sizing servers accordingly going forward. Essentially derivations of Little's law that we all know and love.

Most organizational distributed platforms require software capacity improvements to absorb periodic long latency for downstream platforms while minimizing contribution to latency. This means optimizing thread pools, performing JVM optimization, and understanding thread to CPU utilization correlation based on Little's law. This is how we do it, but do not expect this to be in a manual anywhere in the industry.

Cloud services despite expansive abilities, the cloud is not an unlimited resource. Many times, people have a preconceived notation that scale is unlimited and to just set scaling very high and not worrying about capacity planning. Every service and capability you utilize comes at a cost, including logging, poor error handling, and not doing the proper optimization. Without proper management, these expenses can quickly escalate.

Always consider the efficient allocation of resources. Not all applications need the same level of resources all the time, and understanding this can help in optimizing spending. Sometimes, even limits in Kubernetes will be unset to allow nodes to dynamically allocate what they need. We recommend that you do not do this but instead, set limits to 2x request levels as a practice if you have a lot of variation. Always leverage load testing and chaos engineering techniques to cause more load on one service or set of services to understand behavior and resource availability.

When it comes to planning and setting expectations for third parties using your services or application, the expected downtime should not be zero if you want to keep costs lower; over-allocation of resources for uptime can lead to unnecessary costs. Allow yourself a maintenance window as a practice as well, it is just smart business.

Instead of setting horizontal scaling to a much higher level than needed, implementing practices like bandwidth or request throttling can significantly influence performance

under constrained conditions, ensuring that you are not overpaying for underused bandwidth. This can give you a clearer picture of resource needs and prevent provisioning more than necessary.

Organizations should regularly review their cloud infrastructure utilization and runtime costs to avoid overspending on capabilities that are not being fully leveraged. It is important to not just focus on rapid building of products and having an isolated view of build costs while ignoring runtime costs that is an ongoing expense. Ultimately, taking extra time in the build to reduce the ongoing cost is a smart practice. Optimize the software as it is developed and make sure best practices are being followed around performance. A well performing platform in most cases is a well-designed platform with performance being considered upfront. Managing capacity entails understanding the actual demand and scaling resources accordingly to balance capacity and cost efficiently. By doing so, your cloud services can remain both economical and effective, supporting robust performance without straining your budget.

Understanding third-party dependencies

The following is the process to tune the web thread connection pools for any container to absorb backups from Downstream platforms or other tiers:

1. Determine the maximum response time latency for a JVM or Web Server over the past 12 months using an APM tool (Dynatrace OneAgent, Datadog, AppDynamics, etcetera) for a downstream event in seconds.
2. Determine for how long that peak latency latency occurred (peak event latency duration in minutes) for a downstream event where things backed up.
3. Determine how many connection web threads are being used for a JVM at the worst-case latency time frame.
4. Determine the overall average duration of a web processing thread request during the worst-case latency time.
5. Determine the worst-case peak CPU time for a single thread over the past 12 months.

Then calculate your container pool size to absorb the load allowing for incoming processing requests to still be fulfilled while backup threads are still in-work. You do not want thread suspensions occurring and you want to continue processing during the event.

*Container Pool Size = (Threads Per Second Arrival Rate * 60) * Peak Spike Event Duration in Minutes / (Thread Completion Rate per minute * Peak Spike Event Duration in Minutes)*

Note: Thread completion rate per minute = 60/Max response time latency during backup in seconds.

Then determine how the pool size impacts the overall CPU capacity to determine if more hardware is needed. Order as needed.

*Peak CPU utilization percentage for process at pool size max capacity = (100/60) / ((Peak CPU time in milliseconds for a thread over the past 12 months * Pool Size) * (100/60))*

Network tuning

After you have this information, you can generally build in some additional buffer by tuning a network. This can be challenging but take the approach of math and measure like we do for capacity planning. Normally, on a Linux server, we use the below configuration as a good starting point where applications need high throughput. This is hard information to find so we are putting it here in this book to make your life easier. Here are some starting point attributes to tune for the best performance over the network with IPv4, but consider the dynamics of the network being used or start here and adjust through testing for max throughput and minimal packet re-transmits. Measure and adjust these for the best throughput using real-time transaction loads and a large stream load like a large batch file to stream over the network for batch processing. When you find the equilibrium between real-time and batch settings, you have achieved optimization.

Linux network OS settings for high performance:

```
net.core.somaxconn=4096.
net.core.netdev_max_backlog = 256000
net.core.optmem_max = 100000000
fs.file-max = 2,256,000
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_keepalive_time = 7200
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_intvl = 75
net.core.rmem_default = 33554432
net.core.rmem_max = 33554432
net.core.wmem_default = 33554432
net.core.wmem_max = 33554432
net.core.netdev_max_backlog = 500000
net.core.optmem_max = 100000000.
net.ipv4.tcp_mem = 33554432 33554432 33554432
net.ipv4.udp_mem = 65536 131072 262144
net.ipv4.neigh.default.gc_interval=2000000 (standard)
net.ipv4.tcp_rmem = 8192 174760 33554432
net.ipv4.udp_rmem_min = 16384
net.ipv4.tcp_wmem = 8192 174760 33554432
net.ipv4.udp_wmem_min = 16384
```

Beyond tuning network resources, the same can be said for memory faults in an application. You want to tune for minimal memory page faults. Tuning JVMs is an art in

itself which we will go in depth about in a later chapter. But there is a lot of information on the web about this topic. One thing to remember is that there are two heaps in most JVM versions, more heap memory is not always better in most cases and you want to use multi-threading garbage collection to cleanup both heaps. We see many folks considering the Young Generation cleanup using parallel techniques but leveraging the old serial collection techniques for the Old Generation. This works until you get a traffic burst and the garbage collector cannot keep up. The response times will soar and CPU will be pegged by the garbage collector trying to clean up objects in a single threaded fashion. Spread garbage collection across multiple cores when you can and leverage parallel cleanup across both heaps. When it comes to tuning memory do not tune based on an **application performance monitoring** (**APM**) tool, leverage the native GC logs so you see all of the events. For containers in the cloud, we would recommend leveraging Java 17, which has >20% performance improvement over Java 8 and where you can leverage the optimized memory setting *-XX: +UseContainerSupport.*

Playbook for performance best practices

There are a lot of dynamics to consider to achieve great performance and capacity. We have discussed some of them, and we could list off rules to follow that would be a thousand pages long if we wanted this book to weigh twenty pounds. But, to help, the following are some favorites that will really push your performance and capacity to a new level. Stay away from the anti-patterns listed here and follow the best practices recommended and your platform should achieve great performance.

Excessive layering anti-pattern should not exist in applications. For example, if the application design has grown over the usage of controllers, commands and facades, and these facades have now become the norm. In order to decouple each layer, the software developers tend to add additional facades at each of the tiers. Now, for every request at the web tier, the request call goes through multiple abstraction layers just to fetch the results. Imagine doing this for thousands of requests coming in and the load the JVM needs to handle to process these requests. Too many facades slow things down! Life is a balance and so is software design. Remember when the spring framework was released, and everyone started using metadata for everything, making things non-maintainable? You want to always balance performance with layers of abstraction and latency additions in any software design.

The following are some good practices if you want to achieve great performance.

- Request counts across the network initiated from a web page should always be < 100
- All desktop page **Largest Contentful Paint** (**LCP)** load times should be <= 2.5 seconds on an organizations network connection
- All mobile page LCP load times <= 3 seconds on 4G

- Always use async logging to a FIFO cache in front of disk versus the disk directly
- Always use connection pools, when possible to have active connections and to control load to the database or service
- Always use pagination to minimize request sizes across the network and to minimize scrolling
- Compress service data payloads across the network > 5kbytes in size
- No individual element on a web page should ever exceed 300ms download time
- In web pages, content > 5kbytes in size should be compressed across the network and compression level on web servers or reverse proxies (for example, Nginx, Apache) should be set to max compression versus defaults
- Web Page Weight should never exceed more than the median page weight seen in HttpArchive's State of the Web Report for the top 10000 sites. **https://httpArchive.org**.
- RESTful web requests should respond <= 100ms (processing time) for data retrieval
- RESTful web requests should respond <= 1 second (processing time) for data updates
- Individual downloaded artifacts across the wire should not exceed 300kbytes
- All JavaScript should be minified. Small JavaScript snippets that call across the network to be downloaded should be combined and downloaded all at once
- DOM size should not exceed 1000 elements
- All CSS should be minified
- A single Web Page should not have more than one-page redirect
- All web page responses, dynamic and static, should have a cache response header established
- All non-compressed mime types/content should be compressed across the wire (JSON, JavaScript, CSS, etcetera)
- JavaScript or CSS scripts that are not used should not be included on the page
- Web pages with large amounts of small images < 10k in size should use CSS sprites or other combining techniques to minimize requests across the wire. Always shoot for one request versus many for these situations
- JavaScript should be async/deferred where synchronous dependencies do not exist
- JavaScript files should be combined to minimize requests across the wire
- CSS files should be combined to minimize requests across the wire

- HTML should be minified or at a minimum, spaces should not exist between line items to reduce file size
- Expires headers should be included in all responses and static assets should be cached in the browser. Fingerprinting assets (naming) and storing them for a long period of time (30 days) versus using ETags is recommended
- Stylesheets should use Link and exist in the **<HEAD>** of the webpage to prevent blocking
- Responsive images should have a size specified in the HTML with an image width and height specified. This avoids unnecessary repaints during rendering
- No 404 HTTP errors should exist in the application introducing latencies
- No JavaScript console errors should be present
- No invalid errors should exist in the Logs for the applications
- Exceptions should not be swallowed in Java source code
- Logging information should be actionable and not excessively exceeding 1Mbytes in size each day for a product. This is out of control today which impacts CPU cycles, disk and the capacity of the platform
- Dynamic requests across the network should be cacheable
- CSS expressions should only be used as a last resort and if performance has been verified to not impact mobile or desktop rendering times
- CDNs should be leveraged for external facing applications where heavy use of static assets is used
- Images should be lossy optimized below visual detection to a minimum of 80% quality to minimize image file size coming across the wire
- Web threads minimum values for containers should align with current nominal traffic patterns
- Web thread maximum pool sizes should be set to the level of what the VM can support for incoming traffic or aligned with incoming capacity pipeline projections
- Throttles on all containers should not exceed the capacity of the VM
- Prepared SQL Caches, when available should be set in the container and matched to SQL statement counts in the code
- For web application components which hitting a database directly through embedded components or scripting, the database pool size minimum should be set to nominal load patterns and database pool size maximum should be set peak load levels

- For fast RESTful or SOAP web services min/max DB pool sizes against and RDBMS should be the same (set to peak load pattern) for best performance
- Timeouts should exist at the container for database requests and in the database itself to prevent orphaned threads and processes
- Indexing should exist on all fields being queried for lookups
- The chatty anti-pattern should not exist in applications—With the advent of ORM mappings, Session/DAO objects, programmers started making calls to beans for every data element. This leads to excessive calls between the layers. Another side issue is the number of methods called at each layer that need to start having to support this model. The worst-case scenario is when the beans are web service based. Client tier making multiple web service calls within a single user request has a direct impact on the application performance. To reduce the round tripping, the application needs to handle or combine multiple requests at the business tier. Mapping frameworks like hibernate are not your friend when it comes to performance. Instead of placing an entire domain object model graph in memory with chatting requests trans-versing the taxonomy against a datastore, consider using a transaction script type of pattern when a result set abstraction object retrieves the full result set specific to the request. This prevents the chattiness. Then cache this using something like EHcache, Memcached frameworks or other. If you want things to be really fast consider leveraging stored procedures, but do not put the business logic in there as a good design practice.
- Global data used across multiple requests should always be cached
- Compute Statistics for an RDBMS like *Oracle* should be calculated on a recurring frequency dependent on data growth rate for databases
- Garbage collection should be occurring across multiple processors on each VM to minimize stopping the world
- The Pig Session stuffed anti-pattern should not exist in applications—Session memory should not be overstuffed. Do not be a Pig with your software design. Session objects are a feature provided by the container to track user sessions during the website visit. The application starts with the promise of putting very minimal information in the session but over a period of time, the session object keeps on growing and growing. Too much of data or wrong kind of data is stuffed into the session object for convenience is a problem. Large data objects will mean that the objects placed in the session will linger on till the session object is destroyed. This impacts the number of user's that can be served by the application server node. Many times, applications will use session clustering to support availability requirements but add significant overheads to the network traffic and the ability of the application to handle a higher number of users. Applications servers will see significant performance degradation. To unstuff the session object, take an inventory of what all goes there, see what is necessary, what objects can be

defaulted to request scope. For others, remove the objects from the session when their usage is over

- Everything is a service anti-pattern (everything is a service) should not exist. With the advent of SOA and concepts around micro-services in recent years and the ease of service creation with NodeJS, Golang, and Spring Boot, there is tendency to expose everything as a service. In the older applications, one can observe a similar pattern being implemented with **Enterprise Java Beans** (**EJB**). This pattern, coupled with the bottom-up design approach at times, means exposing each and every data entity as a business service. This kind of granular design might work correctly and functionally, but from the performance and maintenance point of view, it soon becomes a night mare and performance hog. Every web service call adds overhead in terms of data serialization and deserialization. At times, the data (XML) being passed with web service calls is also huge, leading to performance issues when using SOAP or REST protocols. Though JSON is smaller and better, too many services at too granular a level hurts performance. The usage of services and proper implementation of micro-services should to be evaluated from application usage perspective. Micro-services are not good for performance response times overall (though they are for scale elasticity and capacity), so attention needs to be paid to the contract design for balance
- Slow concatenations, applications should leverage **`StringBuffer`** or **`StringBuilder`** classes versus **`+`** operators for more than one concatenation in code
- Long running serial processes should utilize parallel threads
- Chatty services anti-pattern should not exist. If an application has too many fine-grained services, then it is an indication towards the application of this anti-pattern
- Thread-safety synchronization should be used on small blocks of shared resources vs. large scope slowing down applications
- Static long-term information for web applications should be cached at the browser vs. the CDN or web server
- Reconnect, prefetch, preload browser capabilities should be used when feasible for subsequent pages to help speed load times
- Testing should consistent of regular SOAK tests, prod level tests, and breakpoint tests (not necessarily at each release cycle). Example: prod level tests are always required for small incremental releases, SOAK/break/capacity tests on a recurring frequency to understand the platform scale and to look for memory leaks
- Databases should be sharded/partitioned/optimized/denormalized, etcetera to meet application response time requirements
- Error rate should not exceed 5% for requests and 2% is a better target

- Machines being used for testing should be in identical configuration as production environment with respect to hardware, software, data sizing, etcetera or extrapolation models should be defined to determine differences to understand true capacity and performance impacts
- Application or service should be setup monitored in an APM tool with event monitoring
- A capacity strategy should exist for anticipated incoming pipeline along with a projection model
- REST Service responses should use JSON vs. XML to minimize data transfer size across the wire
- Performance test results should reflect transaction mix and usage pattern in production
- Mock services should reflect performance latency variance in production (Lisa/WireMock, other). WireMock does an excellent job allowing configuration a good Galton (log normal) response time distribution curve which means you can stand-in components in place of real components and get a realistic performance pattern for load testing. Most mocking frameworks do not support performance testing well, WireMock is an exception, it does!
- **`ArrayDequeue`** classes or enhanced performance collection classes should be used for FIFO operations or caching with long transversal times
- Long running processes should embrace asynchronous calls vs. synchronous tying up threads
- Integration requests should thread pool outgoing requests to prevent overloading of 3rd party servers, causing issues with long response times.
- All containers should have timeouts specified for each request
- All containers should have Keep Alive enabled for reused connections to services
- GC latency should not exceed 500ms in duration for web pages or 300ms for services. Leverage ZGC or low pause collectors to get this down further and the expense of more compute resources. It will give a better user experience and processing rate
- CPU/Disk/Memory should not exceed 80% utilization on a VM
- Balancing traffic across containers should see response time within 10% variance
- When using container managers like Kubernetes, pod configurations should be at least 25 percent larger than the max JVM heap size. But calculate this size based on off-heap memory utilization, off heap memory needs of the application and any observability tools included. Remember to consider thread stacks and code caches

Scaling cloud architectures

We have talked about this topic some already but it is crucial to have a comprehensive understanding of what scalability means and how this understanding relates to the cloud. In the simplest terms, scalability refers to the ability of a system, network, or process to handle a growing amount of work or to be readily enlarged to accommodate that growth. For a cloud environment, scalability is the capability to handle more workloads without impacting the performance. That can sometimes be the greatest challenge is to scale more workload without large outliers in response times for a rate of traffic coming into an application. We will discuss this further once we go over some of the basics. There are two primary methods of scaling in the cloud, vertical and horizontal scaling:

- Vertical scaling
- Horizontal scaling

Both methods have their benefits and drawbacks, and the optimal choice often depends on the specific needs and resources of your application. But, before delving into the complex world of scaling a cloud environment, let us establish a foundation by first understanding what micro-services are because most IT platforms leverage micro-services that want to leverage continuous integration and deployment scenarios. Let us also talk about how using something referred to in most manuals as **continuous integration** (**CI**)**/continuous deployment** (**CD**) becomes important in cloud platforms.

CI and CD are vital practices in modern application development and deployment. They are particularly important when dealing with micro-services and cloud environments, as they can dramatically improve the scalability and reliability of your application.

CI, often referred to as CI, is a development practice that involves developers merging their changes back to the main branch as often as possible, usually several times a day. This approach minimizes the risk of conflicting changes and allows for quicker detection and resolution of bugs. Automated testing tools are typically used in CI environments to ensure that new code does not break the existing application.

CD, or CD, takes CI a step further by automating the deployment of your application to staging, testing, and production environments. Any changes made on the source code are automatically tested, built, and deployed. The key benefit of CD is that it enables developers to get immediate feedback on the impact of their changes, ensuring that any issues can be addressed as soon as they are detected. Understanding both CI and CD is essential in leveraging the capabilities of micro-services, especially when it comes to scalability within a cloud environment.

Micro-services

In the modern development landscape, a micro-service is akin to a building block and it is used heavily with CI/CD. It refers to a small, independent piece of software that forms

part of a larger system or application. Dissimilar to traditional monolithic architectures, where all elements are interconnected and interdependent, micro-services are organized around business capabilities; each service focusing on a specific function and operating independently from the others. Imagine the process of building a skyscraper. Instead of using one massive brick to construct the entire building, you would apply many smaller bricks, each contributing to the overall structure. This architecture provides immense flexibility, scalability, and robustness, making it a popular choice for complex, evolving applications.

What are the rules around micro-services?

- **Independent**: Each micro-service can be developed, deployed, operated, and scaled without affecting the functionality of others.
- **Specialized**: A micro-service targets a specific business capability, enabling an Agile and targeted approach to development.
- **Scalable**: In the cloud environment, each micro-service can be scaled individually to meet demand.

However, with such flexibility and independence come challenges, particularly when scaling within a cloud environment. In the following sections, we will explore how to effectively scale your cloud environment both horizontally and vertically to suit the needs of your IT platform leveraging an **event-driven architecture** (**EDA**) and micro-services.

An event-driven cloud architecture, or EDA, is a model where operations and services respond to discrete events. These events are significant changes in state, such as a new file being uploaded, a click on a website, or a system starting up. In essence, event-driven cloud architectures allow different parts of a system to react to events in real-time, facilitating real-time, asynchronous processing and limiting the need for continuous polling and unnecessary processing. These go hand in hand with micro-services where one micro-service can respond to an event of another.

What are the characteristics of an EDA?

- **Real-time**: Applications are aware of events as they occur and can immediately respond.
- **Asynchronous**: Applications can process events as they come, without having to wait for previous events to be completed.
- **Limited polling**: Instead of continually checking for changes, the system reacts to state changes as they occur. This is both more efficient and results in timely responses.

This architecture makes use of micro-services, these small, independent applications that can work in isolation or concert, depending on the needs of the IT platform. By using micro-services, systems can be more modular, scalable, and resilient. These micro-services

can be efficiently deployed to Kubernetes, a platform designed to manage something called pods, which are where applications and micro-services run. Kubernetes helps in managing availability and scaling which we will get into further in this chapter. The goal is to be able to scale our cloud architecture effectively, both horizontally by adding more micro-service instances, giving us more scale, and when required, vertically by boosting the capacity of the existing services to handle the ebb and flow of events. Micro-services, running in containers in these pods, offer a high degree of modularity. This makes it possible to adjust services independently and scale based on the requirements of specific tasks. When these services are deployed on a platform like Kubernetes, you get the added benefit of automated scaling, which can be optimized for great scale and performance.

Kubernetes, colloquially referred to as **K8s**, is an open-source platform designed to automate deploying, scaling, and operating application containers. Its primary use case is to manage services and distributions across different machines. The ultimate objective of Kubernetes is to remove much of the manual processes involved in deploying and scaling containerized applications. Why use Kubernetes? Kubernetes provides numerous benefits when it comes to performance and scaling, particularly for applications following an EDA pattern and micro-services.

- **Automated rollouts and rollbacks**: Kubernetes has the ability to roll out changes to your application or its configuration, while monitoring application health to ensure it does not kill all your instances at the same time. If something goes wrong, Kubernetes will roll back the change for you.
- **Service discovery and load balancing**: Kubernetes can expose a container using the DNS name or their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic to keep the deployment stable.
- **Auto-scaling**: Based on CPU usage or other application-provided metrics, Kubernetes is able to automatically scale applications to meet their current demands. This is particularly critical in a cloud environment where efficiency and resource usage can significantly affect costs.
- **Storage orchestration**: Kubernetes allows you to automatically mount a storage system of your choice. It could be local storage, a public cloud provider such as GCP or AWS, or perhaps a network storage system such as NFS, iSCSI, or Gluster.

While our focus in this chapter is on scaling, it is important to understand that Kubernetes is not just used for scaling, but also for managing the state of the cluster, performing workloads, and ensuring the system's overall health.

The downside to using micro-services is that they do not support data consistency very well; instead of having a relational database with all of the relationships, many times tables are aligned with services, and consistency and correlation require multiple database transactions to be managed through an abstraction layer of these micro-services. This is

one of the largest issues with micro-services. Make sure you design or have a transaction management layer on top of the services to be able to set constraints and rollback transactions as needed. Otherwise, you may be in for a great deal of support costs around data issues.

Scaling databases

If you want to ensure great performance, you want a fast database that can scale either horizontally or vertically, depending on the application. You also want to choose a database that can support approaches to reduce calls across the network like stored procedures. Postgres is a good relational database that operates well in the cloud, and MongoDB is a great NoSQL database that operates great in the cloud, many organizations have embraced the flexibility and cost models for these solutions. But these are just two examples. It is always a good idea to leverage **https://db-engines.com/** to understand what organizations are using for different storage types of data. If you stick with databases at the top of rankings around popularity, you can almost guarantee good third-party support.

When it comes to the question of scaling databases in Kubernetes, things can get a bit complicated sometimes. Kubernetes does offer some ways to handle this critical task. One way is through **StatefulSets,** which is a Kubernetes feature designed for stateful applications like databases or any service that needs persistent storage. Kubernetes StatefulSets allow for using stable and unique network identifiers, stable and persistent storage, and ordered and graceful deployment and scaling. With StatefulSets, you can scale out (add more replicas of your database) to handle more demand, or scale in (remove some replicas) when demand decreases.

While scaling databases vertically, by stacking up on more CPU, memory, or storage, is still largely manual, Kubernetes shines through when it comes to horizontal scalability. You can automatically add or remove replicas based on the demand, ensuring your EDA application continues to function at peak performance even as demands change. The inherent flexibility of Kubernetes, combined with its robust toolset for managing and scaling applications, makes it a great fit for building scalable, efficient EDA applications for large IT platforms.

In essence, Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. But how to fully scale pods in Kubernetes? Scaling in Kubernetes is an essential functionality for ensuring efficient application deployment and superior user experience.

Let us explore the key aspects of using Kubernetes in a cloud environment:

- **Autoscaling**:
    - **Vertical Pod Autoscaler**: This feature examines the resource utilization of a pod and modulates its CPU and memory accordingly. This ensures that

the pod has enough resources to function efficiently without wasting the infrastructure's potential.

    - **Horizontal Pod Autoscaler**: Looking at the CPU usage or custom metrics, it automatically scales the number of pods in a replication controller, deployment, replica set, or stateful set based on observed utilization. This is frequently used to scale small micro-services and can showcase a predictable performance pattern if you understand the loss around pod management and balancing across the pods themselves through measurement.

- **Service discovery and load balancing**: Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container becomes high, Kubernetes is able to balance the load and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**: Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Self-healing**: Kubernetes automatically replaces and reschedules containers that fail, kills containers that do not respond to your user-defined health check, and does not advertise them to clients until they are ready to serve.
- **Secret and configuration management**: Kubernetes is capable of managing sensitive information like passwords, OAuth tokens, and SSH keys. It allows you to deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration. Remember, scaling with Kubernetes is not just about increasing resources, it is about efficient allocation, distribution, and management to create a seamless event-driven cloud environment.

Let us go deeper into scaling, which is what this chapter is all about. Scaling can play a significant role when it comes to the efficient performance of applications in the cloud environment. These scaling operations can range from adjusting the size of database instances to increasing the number of replicate servers in your infrastructure. We will focus on how to scale the cloud environment, specifically for EDA applications using micro-services deployed to Kubernetes. Let us begin dwelling on the details. The two pervading types of scaling present in the cloud environment are vertical scaling and horizontal scaling, as discussed previously:

- **Vertical scaling**: Often referred to as **scaling up**, vertical scaling involves increasing the capacity of a single server or database instance. For example, this could entail boosting the CPU, adding more RAM, or increasing the disk space.
- **Horizontal scaling**: Also known as **scaling out**, horizontal scaling manifests in the addition of more servers or instances to your infrastructure. This can effectively distribute the workload and reduce the risk of single points of failure.

When it comes to Kubernetes, there are some handy commands that you can specify to increase capacity on the fly, and they are important to know. Read the Kubernetes documentation around kubectl. For example:

Horizontal scaling command	Using the kubectl scale command, you can modify the number of replicas quickly and easily.
Vertical scaling command	Vertical Pod Autoscaler, it is possible to automatically adjust the quantity of CPU and memory requested by the Pod with kubectl.

Table 4.1: *Kubernetes documentation around kubectl*

Example Kubernetes configuration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 10
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:1.0
        resources:
          requests:
            cpu: "1000m"
```

This configuration deploys ten replicas of **`my-app`** and requests 1000m of CPU for each pod. By modifying the **`replicas`** and **`requests.cpu`** values, you can scale the pods vertically and horizontally. To provide an example of a Kubernetes-based EDA utilizing micro-services, imagine you have an e-commerce application that sells healthy food.

How would we start to architect such an application? Well, first we would break out the different expected high-level operations like user registration, order placement, payment processing, and notifications. These would all be handled by separate micro-services. These micro-services would run in pods managed by Kubernetes and we would use events to

sub, etc). Each event (i.e., new user registration, new order placement, etc.) triggers the relevant micro-service to perform the necessary operations. The key components of the architecture are what we have already talked about above. Once the proper separation of concerns is in place around the domain, then we do the detailed design for each micro-service and assure that it is completely independent.

- **Kubernetes**: This is the primary platform hosting the micro-services. It automates the deployment, scaling, and management of the containerized micro-services.
- **Micro-services**: Each component of the application (user registration, order processing, payment system, etc) is a self-contained service within the system.
- **Event broker**: This is the intermediary that manages the communication between micro-services. It receives events from a service and routes them to relevant services that have registered to react to such events. An event broker can be something as simple as an HTTP server, a Cloud Pub/Sub solution or as complex as a Kafka broker.

Here is how we would break our components as an example:

Function	Micro-service	Events
User Registration	Registration Service	New User Registered
Order Processing	Order Service	New Order Placed
Payment System	Payment Service	Payment Processed
Notifications	Notifications	Notification Send Type

Table 4.2: *Breakup of the components*

Implementing a flexible event-driven architecture within the cloud using micro-services on Kubernetes encourages scalability in both horizontal and vertical directions. Furthermore, incorporating robust database solutions also plays a crucial part in catering to high loads without losing performance or consistency. This is a basic introduction to how Kubernetes, EDA, and micro-services on a simplistic level to begin creating a scalable cloud environment. This is what you need to know to get started.

Now you have some introduction to what application-level scaling is and the types. But what about database scaling? For stateless applications like modern micro-services architectures, scaling is relatively straightforward. But for databases, which are stateful, it is a little more difficult. But just like services running in pods inside of Kubernetes, you can scale databases vertically and horizontally as well, each strategy coming with its own unique trade-offs. These strategies can be divided into the following:

- **Vertical scaling**: Also known as scaling up, it implies enhancing the capacity of the existing database server, such as upgrading the hardware.

- **Horizontal scaling:** Also known as scaling out, it involves adding more servers that work together to manage the database, essentially spreading the load.

Keep in mind that while both strategies serve the purpose, the choice largely depends on the specific needs and constraints of your project. Following a clear understanding and strategic implementation of scalability can streamline your business processes and maintain pace with change. Remember, there is no **one-size-fits-all** strategy. What works best for you depends on your specific requirements. When it comes to scaling databases in the cloud, there are multiple strategies you can employ:

- **Read replicas**: Replicating your primary database across multiple read replicas helps distribute the read load. This can significantly improve performance.
- **Partitioning**: This involves splitting a database into several parts based on certain rules, allowing for more efficient data management.
- **Sharding**: An advanced form of partitioning, sharding splits the database across multiple servers, leading to substantial scaling benefits.

The choice of strategy will depend on the nature of your application, the database technology you are using, your cloud provider, and the specific scaling challenges you are facing.

Embracing a scalable cloud environment can be the difference between an application that merely survives and one that truly thrives. With the right knowledge and tools, Kubernetes technical proficiency is well within your grasp. Inefficient scalability becomes a dangerous pitfall in today's rapid digital landscape. Understanding how to upscale your cloud environment strategically, specifically for EDA applications comprising micro-services deployed to Kubernetes, is pivotal. One of the most important features around high-performance distributed architectures is caching. The most popular places to cache in a distributed web-based platform are as follows:

- Web browser (Expiry headers in responses determine duration, other mechanisms are HTML 5 persistence features and service worker solutions that are prominent in the industry).
- CDN/reverse proxies, web servers, etc.
- API gateway; most gateways like Apigee in the cloud allow caching where you can persist objects your proxy requires across multiple request/response sessions. You can also cache the response of a backend resource. Response caching is especially helpful when backend data is updated only periodically. This is important for performance because response caching reduces calls to backend data sources.
- Global or distributed cache; Redis, Memcached, EHcache server, etcetera are examples. These are used for common data needed across multiple JVMs. It is important to remember that calls to distributed caches make calls across the network, which translates to reductions in scale and slower performance. Caching

of calls. Remember to pay special attention to optimizing garbage collection for the JVM when it comes to large caches. You do not want full garbage collection cycles for large heap sizes impacting the cache performance.

- Localized application cache (Caching content in-memory or on disk with the JVM). Small caches of content to eliminate requests across the wire. Caching data in a localized JVM will always be much faster than hitting a global or distributed cache but remember that it means replicating data when JVMs are horizontally scaled. You want ideally do it for data that does not change very often so that you are not encountering a lot of latency notifying other JVMs when a cache changes. Data like country states or provinces, for example. These do not change very often and do not take up much room. Caching these is preferable over going across the network to retrieve them frequently.
- Database caching; leverage NoSQL solutions that leverage in-memory data storage to assure fast retrieval. For relational databases, precompiling SQL or buffer caching for frequently requested is important. Prepared SQL statements over SQL statements in Java will provide performance benefits. Buffer pools storing most retrieved results, materialized views, etcetera. If using solutions like *Oracle*, this product has something called an **AWR** report that will give insights around internal performance. Many database solutions have a way of obtaining performance insights. Another example is the Spanner cloud database having something called **query insights**.

Optimizing configurations in Kubernetes

It is important to understand how to optimize Kubernetes configurations for scaling. When optimizing containers, you first optimize on one pod. You will set a min pod setting of one and a max pod setting of one. You will tune the pod for optimal throughput and response times with minimal error rates. Leverage tools like LoadRunner, JMH, JMeter or other tools to generate load and take measurements. You will tune the JVM for the lowest garbage collection times under the expected load with the workload model expected as far as traffic going to each method or service. You set the min/max heap size for the JVM based on the expected threads to be used in processing the traffic inbound. You calculate or determine how much off heap memory will be used and consider incoming connection threads and thread stack size settings along with any code cache allocations. Then you take into account any observability agent resource needs leveraging pod memory.

Once the Pod and JVM are tuned, adjusting CPU and memory pod settings to accommodate what is needed, then you multiply the needed throughput by the number of pods needed and assume a lost coefficient of ten percent loss. In other words, if you tune one pod to 50 TPS and you need 500 TPS to support the incoming traffic, you move up max pod setting to 50x *11 = 550*. Why? Because you need to assume loss from balancing requests and overhead. There is always a loss coefficient for any balancing that occurs in a platform. 10% is safe and 10% of 500 is 50.

If you want to be really accurate, measure it by slowly increasing pods and measuring the loss in multiplication factor. Remember, before running a full load test and optimization exercise, you will want to run a single request or 5 concurrently and profile the application and determine how much memory is being used per thread and adjust the memory and CPU settings needed to support this along with the other contributors to resource utilization. You always restrict the inbound thread settings for services to not allow them to overload the JVM. It should not be possible to overload a JVM or pod if you have optimized the platform correctly.

Remember that if you are tuning micro-services to make sure you reach the threshold for horizontally scaling with any thread limits you have put in place and set the threshold at a level where the next horizontal pod can start up fast enough to start receiving traffic before the first one starts slowing down in response time due to reaching too much CPU utilization or I/O bottlenecks. If, for some reason, you have a memory-intensive CPU application, as a last resort, you may need to leverage memory scaling. Remember that the threshold here is set based on the pod memory and your JVM settings for max heap will automatically put you at a high threshold if you do not have your pod memory having enough overhead over the JVM. When Java applications run, they will be constantly going from the min heap size to the max heap size. Consider this dynamic when determining a threshold to set, or before you know it, all pod replicas will be started when you only need one additional one. You need to understand the memory dynamics of your application.

How this works is that Kubernetes will determine the average memory utilization for all pods running and increase or decrease the number of replicas based on the utilization levels against the memory threshold set for scaling. The best way to set this is to run a simple load test against a single pod and run up the load to where you need to scale using observability tools and set the threshold. Do not set this willy-nilly or you will just be incurring a lot of cost and balancing that does not need to happen.

Caching

Caching can sometimes feel complex but remember that every cache hit across an architecture is a request not sent across the network to a backend component. It not only means great speed for the user; it means more capacity capabilities for the platform overall. For example, on a web page perhaps we have one hundred requests going across the wire to the backend web server to retrieve static content along with some dynamic requests to cater to the user, how we cache that data determines how well we scale and can also mean a significant cost reduction for running the platform. Static content rarely changes, which means that it can generally be cached much longer than dynamic content. Where we may have had a hundred requests going across the network originally, after closer examination, we determined that half of the content could be cached locally in the browser. This means that now only fifty requests go across the network to the backend of the platform. From a capacity perspective, the backend platform now only needs to be able to send half of the traffic. Many times, web pages have a lot of static content that rarely changes. This means

we can cache it for long durations. How long to cache is generally based on an expiry policy returned in HTTP header for a web response. When we request data a second time, client app tries to retrieve the data from the browser cache first. Dynamic content can also be cached but for a much shorter duration or for a long duration with a mechanism to determine if something has changed on the backend to warrant bringing down the content.

This is just a simple example and the most common one overlooked in designing distributed architectures with a web component. But in reality, almost every layer has a caching mechanism that should be used for the exact same benefits which multiplies the advantages overall for the platform. *Figure 4.1* showcases a simple example of where caching takes place in a distributed architecture. First, we have the browser cache, which we discussed. Then we have something called a CDN, which will cache content and periodically retrieve data back from the origin server. CDN is a geographically distributed group of servers that work together to provide fast delivery of Internet content. It allows for the quick transfer of assets needed for loading Internet content such as HTML pages, javascript files, stylesheets, images, and videos. The magic of the CDN not only lies in its ability to reduce bandwidth costs and improve website load times, but also in its application of caching. Let us decipher the role of caching in a CDN and how it maximizes a website's overall performance. In the context of a CDN, caching plays a crucial role in delivering content efficiently. A CDN consists of multiple servers distributed across different geographic locations. These servers store cached copies of the website's content. When a user requests a webpage, the CDN determines the closest server to the user and delivers the content from that server's cache. This reduces the latency and improves the overall user experience by minimizing the distance the content needs to travel.

You can also cache at the API gateway, databases, JVM, distributed caches, reference proxies, web servers or micro-frontends, and more; even the balancer at some level. What is most important to consider is how long you should cache based on the type of data being cached, and what cache algorithm you should use, such as LRU/MRU, etc., but never think that you should ever not cache. It may sound complex, but it will be the largest driver to scale. Any request that does not need to go across the network is a saving for capacity and improvement to the user experience. It also means more resilience for the platform when something is not temporarily available on the backend. If you can pull from a local cache for a database response for example and keep that data instead of continually to query the database across multiple layers, it means better capacity and more resiliency if one of those backend components would, for some reason, be unavailable for a short period.

Caching algorithms are the unseen heroes when it comes to performance. These algorithms can be sophisticated and work tirelessly behind the scenes to enhance our experience with an IT platform. They make our online interactions quicker and smoother. This is achieved by storing and managing predicable data in caches for retrieval, reducing the time and resources required to retrieve this data from its original source. Think about how much

of information. What if that information could be cached at a layer closer to where it was needed? Every caching algorithm predicts what data will be needed next and have it readily available. That is the goal at least. Some of the more popular algorithms are:

- **Least recently used** (**LRU**)
- **Most recently used** (**MRU**)
- **Random Replacement** (**RR**)
- **Least frequently used** (**LFU**)
- **Time-to-live** (**TTL**)
- **First in first out** (**FIFO**)
- **Adaptive replacement cache** (**ARC**)

Let us begin our discussion with LRU caching. LRU will seek to prioritize the most seldom used data, discarding it first when space is needed in the cache. Implementing LRU necessitates recording every access, which in and of itself can consume quite a bit of memory, which is one of the downsides. This is a very popular algorithm and you will see it used frequently in IT platforms. Another popular approach to caching is called MRU. This one releases the most recently used items in the cache first. It does this when the cache is full and needs to make room for new data. This is based on the assumption that data accessed recently will not likely be needed in the near future.

RR caching is simple, though not always efficient if not used at the right time. Its simplicity is the value and it is very easy to implement. It discards items randomly when the cache is filled to capacity and requires space for new data. It should only be considered in specific situations, where predicting future data needs is not feasible or possible.

LFU is another algorithm based on access frequency. It records the frequency an object is needed over time, then it learns to discard the cache items not used very frequently when the cache is full. It can be computationally heavy to track frequency is the only downside.

TTL uniquely incorporates time into the caching algorithm. Each cache entry has a time stamp when it is saved, and it is removed once its time limit has been exceeded. You will see this one heavily used throughout large IT platforms.

The FIFO caching algorithm removes the oldest data first, protecting the newer data. This is like a line in a coffee shop. The customer who came in first is served first, and so on. When logging to the slower downstream system or a disk, for example, you typically use a FIFO cache to write to memory and then, after the fact, slowly right to disk to speed up the application. It does not need to wait to write to a slower device and hang up the thread longer than necessary.

Lastly, let us talk about the ARC**.** This is a much more advanced cache than the other ones we have talked about so far. It is complex will dynamically adjust its behavior based on the characteristics of the workload. It records and learns over time to balance

between recent requests and frequency and the longer patterns over time. *Figure 4.1* below represents popular caching points in an IT platform. But remember to always consider it in everything that you do when involved in architecting or a grooming session for an IT platform. A transaction not sent across the network ultimately means more capacity and more performance for the platform. Its complexity early on is well worth the reduced runtime costs and higher performance later.

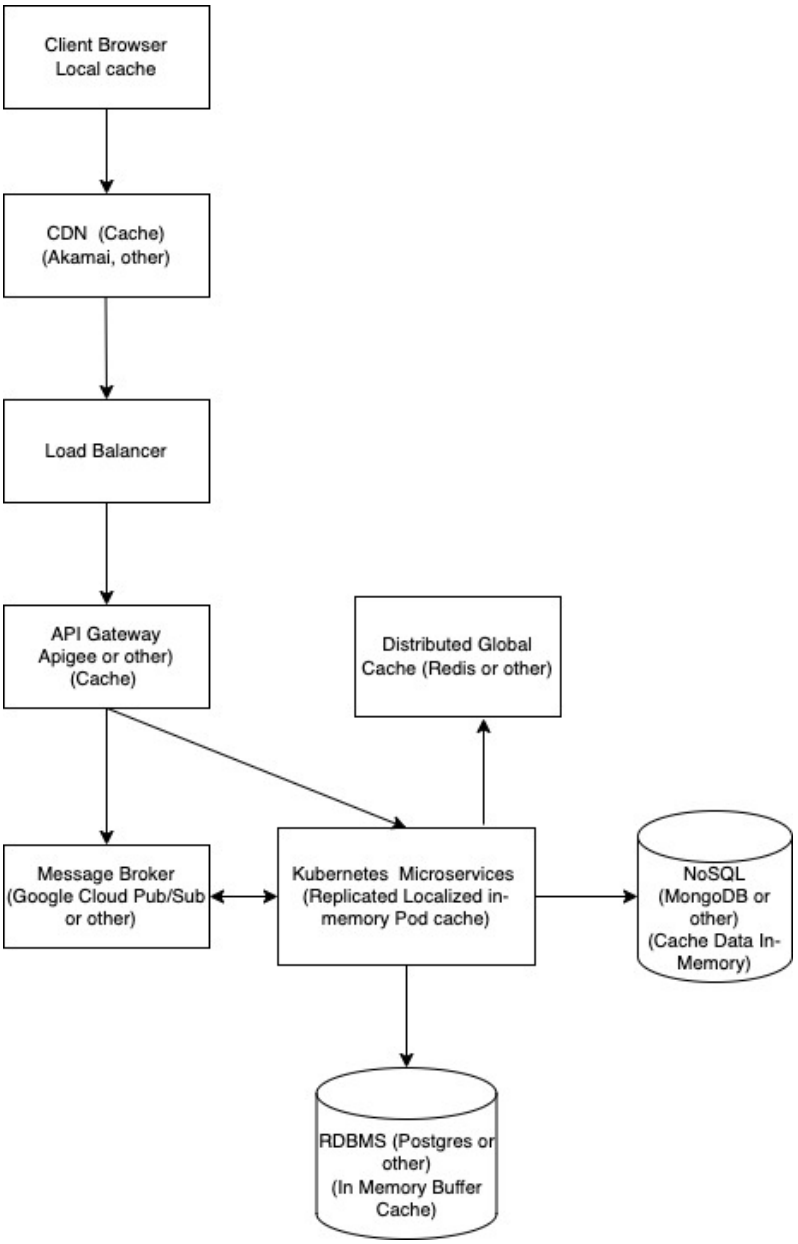


Figure 4.1: *Caching available at different layers*

Conclusion

In conclusion, in this chapter we delved into important concepts in computing; Little's law, Amdahl's law, the USL, and the art of capacity planning in the cloud. We talked about caching, event driven architectures, micro-services, Kubernetes, and much more. We hope this chapter provided you insightful knowledge and considerations into these core theories which underpin efficient cloud computing. Understanding the laws around capacity planning and cloud computing is extremely practical. Remember to set quotas (scale limits to control costs) in your cloud software to make sure that teams adhere to limits set, this encourages optimization and lower costs.

In the next chapter, we delve into important concepts of performance automation to assure that platforms can be delivered at high velocity with high performance quality. We will also discuss more about quality gating concepts. Understanding performance automation in cloud computing is extremely practical including automatic gating. It helps with determining bottlenecks and driving optimization work that needs to be completed to assure the customer has a great experience and that the platform continues to have great capacity as new versions of the software is released.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 5
Performance Automation

Introductions

This chapter will introduce you to **performance automation**. It will help you understand the importance around implementing automated performance solutions for an IT platform. In previous chapters, we discussed some of the tools used in automation and we will talk more about those solutions in this chapter. This is not a large chapter, but it is an important one to draw attention to the importance of automation with performance to assure fast delivery with high quality. Too many organizations are doing nothing but manual ad hoc load tests and not following a good performance practice. The result is that their products are slow to deliver and costly to maintain, keeping customers and revenue is challenging. They cannot keep up and balance quality and release velocity. Automation to solve this problem is not well understood or considered. In this chapter, we will discuss some of the challenges and realities around automated performance measurements and discuss metrics to use for quality gating. Refer to previous chapters for code fragments for **Java Microbenchmark Harness** (**JMH**) and Selenium frameworks talked about in this chapter or peruse the web. We discussed a lot about these up to this point and this chapter is about where to use them and why quality gating is important. We hope that you enjoy this delve into automation, but remember that this is just the tip of the iceberg. So many tools and frameworks are available today to automate and monitor your platform like k6 or Playwright. Sitespeed.io is another example of a platform that can showcase some great observability at low cost.

Structure

This chapter covers the following the following topics:

- API and journey level performance automation
- Quality gating

Objectives

In this chapter, we will discuss important concepts of performance automation to assure that platforms can be delivered at high velocity with high performance quality. We will also discuss more about **service level agreements** (**SLA**), **service level objectives** (**SLO**) and **service level indicator** (**SLI**) concepts. We hope this chapter provides you with insightful knowledge into how to ensure your customers receive a quality product which underpins efficient cloud computing, and guides you in effective IT platform delivery. Understanding performance automation in cloud computing is extremely practical. It helps with determining bottlenecks and driving optimization work that needs to be completed to assure the customer has a great experience and that the platform continues to have great capacity. Let us not wait any longer and dive right into it!

API and journey level performance automation

As speed to market becomes more important and release cycles become more frequent with the industry embracement of both **continuous integration** (**CI**) and **continuous delivery/deployment** (**CD**) of software**,** we need full automation around performance to be able to release our IT platform at a high velocity in order to assure high quality. This means using solutions like Jenkins to automate and deploy our code and control the frequency of releases and quality gating.

https://www.jenkins.io/

Jenkins will streamline our development process and provide an efficient solution to invoking performance automation as part of quality gating in our CI/CD software deployment processes. Jenkins is an open-source automation server, it lends itself building, testing against quality gates, and deploying our software. Let us further understand why people across the world use this powerful tool:

- **Continuous integration:** Using Jenkins, we can integrate the changes made in a project in real-time automatically via plugins and observe the results. For example, when we commit a change to software, Jenkins can automatically receive an event and build and invoke performance and functional testing code we have put together in Selenium, JMH and JMeter, easily and record the results of those tests, and if they pass or fail the quality gates that we have established. If it all checks

- **Continuous delivery**: Jenkins has features for continuous delivery where after integration and quality gating, the changes can be automatically sent to production. What this provides us is high velocity delivery where we spend much less time in slow manual tasks and spend more time producing high performance features.

  Using products like Jenkins, productivity increases, and our software quality takes a major step forward. Continually building with performance and functional testing and quality gating are assuring the quality for our IT platform. It means that our software changes are evaluated and only the best version of our software reaches the IT platform's users. When you use Jenkins for building, remember that you want to fail the build if quality gates for performance and functional testing does not pass.

- **Performance testing**: It is crucial to know how our software will perform under concurrency conditions. Jenkins' Performance plugin allows running of tests at each change and provides detailed statistics that can help spot performance degradation. Tools like Dynatrace, JMeter, JMH, all have plugins to Jenkins as well to see a 360-degree view of performance. So do popular testing solutions like LoadRunner and NeoLoad. Remember that JMH was designed as a very low standard deviation micro benchmarking tool with little standard deviation in response times (written in C language) and contributes little to the testing latency itself, unlike some other solutions. When combined with a good web service client, it can be very powerful performance load testing tool with much better repeatability than other solutions and without the need for large amounts of horsepower for load generation. It goes far beyond unit testing and is an excellent solution for API load testing. Solutions that generate load written in C language, Golang or RUST tend to require much less horsepower and cloud infrastructure runtime cost. They also, typically, have less standard deviation and additional latency contribution to tests.

- **Quality gating**: Quality gating involves making decisions based on the quality of code. The SonarQube plugin integrated with Jenkins does this, examining our software with a fine-toothed comb to weed out any potential bugs, code smells, or security vulnerabilities. But the magic when it comes to performance gating is in the establishment of SLOs and having tools like JMeter or JMH test performance to the SLOs in an automated fashion and fail integrations or builds as part of the delivery process when quality gates are not met. Other solutions that are popular or introducing Chaos Monkey for resiliency testing or K6 if using Kubernetes to introduce degradation of the runtime environment. Solutions like NeoLoad have built in SLA capabilities into the product to fail Jenkins builds.

Jenkins not only takes care of our project delivery but ensures that we reliable software every time.

When releases occur frequently, perhaps every two weeks as part of a sprint in Agile processes or monthly even, running detailed performance analysis and measurements for quality gating for every release manually can be very

participant in the world of IT platform development, both making performance quality gating as part of the build process and ongoing continuous performance testing automation is key to our software's success. This chapter will introduce the concepts of performance automation, a milestone in quality gating and improving our software's performance throughout its entire life cycle. We will do quality gating far left in the process all the way through integration testing before releasing it into upper environments. The farther left we can put robust processes, the lower the cost to deliver the platform.

Before doing any performance automation, we need to first establish SLO. But before we go too deep here around requirements and thresholds for quality gating, it is important to understand the differences between different quality gate threshold metrics. You will hear the terms SLO, SLI, and SLA frequently, but what do they really mean?

Service level agreements

Let us talk about the most popular term that has been around the longest, SLA. An SLA is a contract between a provider of services and the end user or another service provider in the case of business-to-business agreements. Many times, when that contract is breached, one party compensates the other party for any revenue loss, including bounce rates (customers leaving while using the IT platform) from websites providing a service.

Understanding MTTR and MTTA

The most popular attributes of this metric are response time and availability. But uptime is many times included as well and sometimes you will see contracts include parameters like **mean time to repair** (**MTTR**) or **mean time to acknowledge** (**MTTA**) in those agreements. MTTR represents both scheduled and unscheduled in many, though not all, cases, and this is a vital metric for evaluating the reliability and efficiency of a service or application. Of course, the quicker the response and repair, the better the metric, with less downtime and better performance. In the past, we have seen MTTR agreements between organizations set a threshold limit around revenue compensation. For example, a high priority fix should be addressed within one hour, if it takes longer, compensation around revenue loss or penalties are ensued. MTTA is self-explanatory and is not used as often in SLA definitions compared to MTTR. It is the average time it takes someone to respond when an alert goes off in an IT platform indicating that something is wrong.

Service level indicators

When we hear employees in organizations use the term SLI, we always question whether or not they truly understand what this means. Generally, what we see is some young performance person will *Google* the web and look for a definition and put it into a presentation to leadership and ultimately this leads to confusion. Most definitions on the web are ambiguous without clarity around this metric. Most vendors make it unclear as well in their documentation. At the end of the performance presentation, there is rarely good

Service level objectives

SLO are what we use to quality gate on primarily with performance automation. This sets the performance level goal we want to achieve to meet an SLA or to assure good capacity or user experience for the customer. It is the threshold to meet for a service to be released into production and availability or resiliency, thus error rate, is also included in the automation for the quality gate. If we start seeing HTTP 404 messages indicating resources cannot be found when retrieving a web page, we have an availability problem and the software should not be released into production. So, think in terms of SLO as error rate and response time thresholds for a given processing rate. For a very simple example in understanding, we will remove all metrics except for response time for clarity, because this can quickly become unwieldy. If an SLA takes 10 seconds for a service to respond, we will generally set an SLO threshold much lower to keep us safe. So, the SLO may be 7 or 8 seconds for a threshold. SLI represents the measurement itself. Perhaps our SLI is 6 seconds and the service is released into production.

We have also seen SLI used in a different way and different understanding in organizations where it represents lower-level measurements that comprise up to an aggregate around the SLO threshold. For example, let us say an SLO is 8 seconds for a web service and that web service relies on database latency being under 2 seconds, disk latency being under 100ms per read operation, and parse latency to transform content from the database to a format recognized by the browser in a readable way of 4 seconds. The term SLI we have seen organizations represent as the measurement of the individual parts required to achieve the high level aggregate SLO. If disk latency goes to 1 second, then the SLO would not be met, for example. But at the end of the day, it is the actual performance measurement or tracking individual measurements over time in the platform to see if that SLO or SLA can truly be met and how that data is tracked. The confusion is typically whether the aggregate measurements are tracked or the individual granular measurements are tracked that determine if the threshold will be met. Sometimes it is both. Something to remember here is that this can quickly become overcomplicated with multiple metrics being tracked and their relationships. In business always remember that simple is always better. It is better to have something simple that an organization understands its goals than making something complicated where only a small few realize what is going on. If you remember the term **keep it simple stupid** (**KISS**), this always applies when it comes to building and tracking performance and resilience of IT platforms.

One of the first questions to ask about performance automation is at what stage in the production creation process do we automate. Here is where to use performance automation and quality gating:

- Component unit testing
- Integration web service level API testing
- Front end journey level testing

Tooling used for performance automation

We will harness the power of JMH for comprehensive unit testing and effective load testing on web services, as well as the remarkable benefits of multithreading the web client for accessing APIs. Unit testing will also be done using JMH. The JMH is a powerful toolkit utilized to create, run, and analyze performance tests in Java and other JVM languages. It is specially engineered to measure performance of services, components, methods and even small code snippets, often referred to as micro-benchmarks for unit testing. It has the ability to generate multi-threaded load for load testing APIs using built-in java web clients or web client testing frameworks like REST Assured as well, which many people do not know about. JMH has been around for decades and has evolved into something special beyond simple unit testing. These days, it can execute load and even do profiling around compute, code cache sizing and garbage collection. Not a lot of information on the web about the advanced features of this framework with most folks thinking that it is just to measure method latencies only in a single measurement.

With a robust framework that mitigates common micro-benchmarking pitfalls, JMH offers precise and accurate results which is what is important. For example, where tools like JMeter may introduce up to 200ms in standard deviation between measurements due to its own overhead, JMH only introduces 2ms and is very repeatable. It is a crucial tool for developers striving for optimal performance in their applications. When selecting the tool to use for your quality gating, do not just consider features; consider repeatability and lowest overhead scenarios. Nothing worse than using a load testing tool that contributes to a great deal of latency itself around what it is reporting.

The same goes for monitoring solutions like **application performance monitoring** (**APM**) tools like Dynatrace or Datadog. We do not want to just focus on features, we want to choose monitoring solutions that generate little overhead themselves. This means solutions that do not introduce more than 200ms of latency. When we get above this threshold, we start doing our platform more harm than good. When companies have a trial product look at agent file sizes, memory consumption in JVMs they are monitoring, and repeatability of measurement. How much does it cost to run this tool and is it going to impact your customer bounce rates because it is slowing down your application? Take a hard look to understand how they are contributing to latency or measurements being seen. Even considering the programming language they were written in, you can almost guarantee a solution like JMeter, written in Java is going to be slower, consume a lot of resources, and contribute to latency more than an application written in C or Rust. The bottom line is to look at everything, do a **proof-of-concept** (**POC**), and profile what is happening. Never rely on the brochure around feature comparisons with competing products.

So, what are some best practices around code performance? For all method level benchmarking in unit testing, we want to choose best practices around SLO for quality gating. Methods should all respond in 100ms or less for read operations and no more than 300ms for write applications, as a simple example. To do this for services interacting with datastores, make sure that all database interactions are

or updates being made to it. Use JMH to benchmark the methods or use a profiler and optimize the code to meet these criteria.

For journey level (user interface) performance automation and quality gating, one good solution is to leverage JMeter with Selenium. Testing tools to combine with JMeter, like Playwright are also great solutions. If you want a solution that will do both performance (web and API) and functional testing of these components along with chaos engineering, k6 is a great solution for this. These tools can revolutionize the way you design and test your web interfaces. But remember no matter the solution, you must be able to quality gate against thresholds set in files or a database representing your SLO. The goal should be, as part of the build process, to run through API tests for performance JMH, and then use JMeter, Selenium, or Playwright, or other solution like k6 with a device cloud for your journey level automation.

It is important to define a SLO and understanding how to do that is many times not understood in organizations today.

If our business does not have a competing product to beat in performance or cannot define the performance needed using it as a guideline, we want to set SLOs to levels better than what HttpArchive.org establishes for the State of the Web report for the median page load time. This will guarantee that we will have a high-performance product that people will love. Once defined we will want our products to track with these metrics over time as expectations change for the web. If we look at the top 10,000 sites or top 1,000 sites on the web, we should see some valuable metrics to shoot for when designing web applications for performance. If you want to see web vitals for a website, you can leverage tools like **Chrome User Experience Report** (**CrUX**) as well. But look at the successful sites in performance and model your requirements after that to keep your customers happy. If we know of a competing product, we need to beat their thresholds around performance, at a minimum.

Device clouds

As part of a comprehensive performance testing strategy, leveraging the power of a device cloud service such as LambdaTest is crucial. Now, there are many of these on the market with BrowserStack, Perfecto, Sauce Labs, with Lambda being the most comprehensive but these solutions leapfrog each other every year. Performance testing should never be constrained to a single device or browser variant. This is the most important thing to remember. You must test the performance across browsers and devices, as it will significantly vary. Knowing which browsers to test with depends on the users of the platform. If the platform is already in production, then using analytics tools can determine which browser customers are using. These are what you would need to target as a requirement. LambdaTest, like these other large device cloud providers, supports an extensive browser and device coverage, thus ensuring that your web application's performance is top-notch across different environments.

LambdaTest enriches your testing process by enabling you to experience how your application performs across a variety of devices and browsers. Here is how it impacts testing:

- **Comprehensive device and browser coverage**: With LambdaTest, we can conduct tests on over 2000+ different browser types and versions, and on a vast array of devices.
- **Simultaneous testing**: LambdaTest allows simultaneous testing on multiple browsers and devices, thus saving valuable time and enhancing testing efficiency.
- **Performance analytics**: Gain qualitative insights from detailed test logs, video recordings, and comprehensive performance reports.

Performance variations across browsers and devices can profoundly impact user experience. Hence, having a robust cross-browser and cross-device testing strategy using LambdaTest or other device cloud is instrumental in optimizing overall performance. Moreover, LambdaTest does not stand alone in the performance testing ecosystem. It integrates seamlessly with tools like Playwright, JMeter and Selenium, enabling you to combine the capabilities of these tools for a more comprehensive performance automation solution end to end. By merging these toolsets, we can build a strong wall of performance assurance that stands strong against every usage scenario your application may encounter. So, why not give it a try? Enhance your performance testing strategy Selenium or Playwright, JMeter, LambdaTest and JMH and see the difference. Remember to use the quality gate using SLOs!

Quality gating

The term quality gating is used throughout this book, but we really have not given a formal definition around this and have been relying on the discussion context regarding how it is used to assure your understanding. However, formally, quality gating is what signifies software readiness. Once SLOs are met, the software is ready to be released. The same goes for quality gating around functional testing. All tests should be passing for the software to be released or the build should fail. When quality gates are not met, it points out where the bottlenecks and platform defects are. These serve as a gatekeeper, evaluating each software build against predefined quality-related criteria before it passes onto the next stage of the software development life cycle. If we start seeing 1 second method latencies in unit testing using JMH, should we allow the software to move into integration testing? The answer is no! We want to ensure that SLOs are met before moving the software to the next stage in the process. Then, at that next stage under load, if we see the SLO not being met, the same thing happens; it never gets to the next stage of the process.

The positive aspects of quality gating are as follows:

- Prevents promotion of slow or non-functional code
-

- Reduces defects in production and ticket backlogs
- Reduces impacts on revenue
- Pushes ownership to the person creating the code to meet objectives
- Reduces alerts and support costs and prevents consuming people's time

The disadvantages of quality gating:

- Slows initial planned release times and velocity
- Initiates unplanned activities around performance

Balancing quality and release velocity is crucial for a great product. No doubt that quality gates can slow down the process of software development. This is because they demand more time and careful scrutiny of code. Moreover, inexperienced developers may struggle to meet the stringent coding standards. This can lead to loss of productivity and may even cause developers to avoid undertaking complex tasks. That being said, code should be developed with high performance and rigor out of the gate, and developers should take ownership of the code. Code should have ownership and full documentation using tools like Javadoc. But shoot for simplicity around requirements and best practices defined, always.

As our platform development moves towards the finish line, our discussion further emphasizes the value of performance automation. When we weave it seamlessly into CI/CD and quality gating, we not only build stronger software but also create better user experiences. So, like before, we cannot hesitate to ramp up our performance testing game with these great practices. If we try to do manual testing in today's world of continual releases, we will never have a high-velocity platform, and we will never be able to keep up with frequent release cycles. To do this, we need automation for everything!

Beyond unit testing with JMH, we need to think about taking our performance automation a notch higher by moving it farther left in the development process. An effective strategy you might want to consider is running Lighthouse Reports in an automated way. These can be generated using tools like Selenium, Playwright or Puppeteer. They can also be run manually from the browser for both mobile and desktop viewport sizes. The **Lighthouse** tool is integral to the *Google Chrome developer tools suite*, providing invaluable insights into the performance of an IT platform's web pages. Remember, performance is not an afterthought. The best practice is to ensure that the platform is on track before moving the software into integration testing. Keeping an eye on the Lighthouse performance score from the beginning can save an organization from unforeseen shocks later in the game. Also, look at the browser capabilities around Performance Insights in the Chrome developers tools suite to analyze webpages with ease around core web vitals.

Core Web Vitals are a set of only three basic metrics implemented by *Google* to evaluate and enhance user experiences on websites. You will see these in modern APM tools that have **Real User Monitoring** (**RUM**) in many cases. If you have worked with performance

engineers in the past, you know what these are. They focus on three major aspects which determine how a website responds. One metric is **Largest Contentful Paint** (**LCP**), which measures how long it takes for a page's main content to load. This is what you set your SLO against in journey level automation in most cases. We used to leverage **time to interactive** (**TTI**), but that metric has become too subjective across different browsers. **First input delay** (**FID**) is the next important vital metric which gauges a user's interaction responsiveness. **Cumulative Layout Shift** (**CLS**) is the last important one that assesses the visual stability of a page to avoid unexpected layout shifts.

Optimizing these aspects is crucial to deliver a high-quality, user-friendly site and can also influence your website's search engine ranking. Using tooling like Lighthouse or Performance Insights built into the Chrome developers tool suite can be instrumental in understanding vitals and how to optimize them for your customers. Remember that you can run Lighthouse programmatically, meaning that you not only run unit tests with JMH in an automated way far left in the process but also for screens being created using Lighthouse. Lighthouse scores can be used as a gating criterion as well for performance and accessibility (uses axe tools by dequeue systems under the covers). Everything you find farther left in the process means cost savings for the production creation. So run JMH against SLOs, and Lighthouse and if performance scores are not met, do not let the software through. When using Chrome Developers tools for performance measurements or tools like Lighthouse always remember to setup profiles that represent the up and down bandwidth that your customer base will be using. So, if an organization has cable customers that have a 100Mbyte download rate and a 20Mbyte upload rate in their connection, we need to be sure to represent that in all measurements, or they will be invalid. Most organizations have a very fast bandwidth speed compared to the customers accessing their product. So, if an organization has a 400Mbyte download speed and a 200Mbyte upload speed and this is what you set your SLO's against, then customers with 100 Mbyte upload with a 20Mbyte download speed going through their **internet service provider** (**ISP**) are going to have a terrible experience. If the performance requirement around page load time being met is 3 seconds, and this is measured in testing with 400Mbyte download speed and our customers have 100Mbytes, they very likely will see 9 seconds vs. the 3 seconds for a response time or more depending on the path and contributing ISP latencies.

Here are some key benefits of utilizing Lighthouse for performance automation:

- **Automated auditing**: Lighthouse provides automated audits for performance, accessibility, and other web vital checks. Integrating these audits in your CI/CD pipeline helps you maintain top-quality delivery standards.
- **Progressive Web Apps (PWA) readiness**: Lighthouse is an impeccable tool to test your PWA readiness. It checks against a pre-defined checklist to help you gauge your PWA's performance.
- **Flexibility and usability**: Lighthouse can be run from the command line or as a **`Node`** module, providing flexibility. Moreover, it can audit any public web page, requiring no special setup.

Lighthouse, when used in combination with other automated performance testing tools like JMH, JMeter, and Selenium, or Playwright, can prove vital in understanding and improving the performance profiles of our software throughout its life cycle. The goal is to maintain consistently good performance scores, helping meet and exceed the defined SLOs.

Chrome developer tools offer a different dimension in acquiring Performance Insights. This extensive tool that has become so popular in the industry provides a wealth of information in regard to page load performance along with runtime execution. It is built directly into *Google Chrome browser*, giving real-time insights into the working of a web application, including understanding requests across the network, profiling performance, understanding bottlenecks, and gauging the impact of front and backend optimizations. Many times, backend request times impact front end rendering times because browsers are limited in threads they use for rendering. When one gets hung up on a backend request, then rendering cannot complete until that thread is available again or the rendering or code execution in the browser becomes slower. Front-end optimization is something we will talk more on in an upcoming chapter. However, to help you get started, here are some of the key aspects of what Chrome can provide for performance analysis. Do not hesitate to use this tool, it is one of the best out there for understanding what is happening for your platform. *Google* has taken a strong interest in performance in recent years and works hard to ensure its browser is fast. How they do that is through the observability built into the browser that they provided themselves.

Key features and capabilities of chrome developer tools include:

- **Loading and scripting performance profiling**: This reveals how much time the browser spends loading resources, scripting, rendering, painting, system related tasks, or idle. Browsers load resources in what you call a waterfall pattern due to the thread limitations in the browser. *Google* can show you this waterfall and allow you to see where resources are queuing or blocked.
- **Identifying forced synchronous layouts**: Synchronous layouts can cause serious performance issues. Chrome developer tools help in identifying instances of these layouts, facilitating their rectification.
- **Flame chart analysis:** The flame chart provides bottleneck insights into the execution of your application Imagine you are in the middle of programming and need to quickly find performance issues. You do this through code profiling and what identifies the issues quickly is a flame chart. A flame chart is essentially a data visualization tool that software developers use to monitor and understand how their programs are running. It consists of different layers, each layer representing a software function, and the width of each layer indicating how long it took to run. This information is crucial for spotting bottlenecks and improving the efficiency of your code. Remember that the next time you find yourself lost in lines of code, remember to profile the code and look at the flame chart.

Conclusion

Performance automation plays an indispensable role in maintaining the efficiency and effectiveness of web pages. In the modern world, users demand quick, responsive, and reliable websites. This means every second matters. Through automation, the complex and multi-faceted task of monitoring and optimizing web performance metrics like Core Web Vitals becomes streamlined and consistent. It allows developers to identify and address potential issues preemptively, preventing them from escalating into larger problems that could impact user experience and search rankings. Furthermore, automation saves valuable time and resources that can be dedicated to enhancing other dimensions of a website. In essence, implementing performance automation is not just important but essential to maintain a competitive edge in the evolving digital landscape.

In the next chapter, we will learn techniques to optimize web platforms operating in the cloud. What we learned in this chapter provided us a great deal of information around using JMH, JMeter and Selenium combined to provide with tools like Jenkins that we can leverage for automation and CI/CD. In this chapter, consider what we have learned so far. Remember that JMH, JMeter, Selenium, Jenkins and a device cloud like LambdaTest can be combined into a powerful tool suite for performance automation and quality gating. That has been the focus of this chapter. But we need to always do our homework and look at other solutions like k6 and Playwright combined with LambdaTest and other device clouds as well. At the end of the day, the message of this chapter should be that testing across multiple devices is important in an automated way with quality gates. It is important to be able to do performance testing and quality gating on both front-end rendering aspects of web and mobile applications as well as backend API testing. It is important to make it a part of the build process and leveraging tools like Jenkins helps being the glue to make this happen. Resiliency testing is also important and using solutions like Chaos Monkey, is a good solution or leveraging K6 if using Kubernetes. We want fully automated measurements around performance, functional and resiliency testing, and quality gates to make sure we pass or fail builds going into production. As an exercise, consider the following questions:

1. What is Jenkins used for?
2. Why is a device cloud used when it comes to performance engineering and functional testing?
3. What is quality gating?
4. How do we define service level objectives?
5. How is Selenium used to measure performance?
6. What is JMH?
7. Here is a harder one. Consider why we cannot just run JMeter without any additional tooling for performance testing? What is JMeter missing?

Answering these questions, means the chapter was well understood and we are ready to

Chapter 6

Cloud Web Platform Optimization

Introduction

This chapter of the book will introduce you to cloud **web platform optimization** (**WPO**). Web platform optimization is an aspect of website development that should never be overlooked. Struggling with slow websites can be frustrating for all of us, but the good news is that it is entirely preventable.

In an age where speed matters to users, understanding performance optimization and implementing strategies for improvement can make a dramatic difference in your online presence.

Note: Remember, a faster and more responsive website not only brings in more user engagement, but it can also improve your SEO rankings.

Structure

This chapter covers the following topics:

- Identifying web performance bottlenecks
- Understanding rendering workflow dynamics
- IFrame page load impacts

Objectives

In this chapter, we will introduce you to the important concepts of WPO in the cloud. We sincerely hope that this chapter provides you insightful knowledge regarding how to assure that your customers will receive a quality and fast product. It involves employing a set of performance strategies and techniques intended to help speed up the delivery of a website's content to browsers and improve interaction times. However, its importance lies in its direct impact on user experience, engagement, and conversion rates. These aspects are all critical for a successful business having an online presence. Let us not wait any longer and dive right into it!

Identifying web performance bottlenecks

Right now, as you are reading this book, people around the world are surfing the web looking for or interacting with different content. They are doing all sorts of things, from reading blogs and articles, watching videos, participating on social media, shopping online, conducting business, and perhaps enjoying the fruits of capitalism. IT web platforms make this possible. But have you thought about how well these platforms are built or how building platforms in different responsive ways impacts the revenue dynamics? Do you ever consider how their fast performance can drastically improve the users experience and what impact that fast performance has to a business?

Understanding and harnessing the potential of WPO becomes exceedingly critical to the success of every IT solution that drives revenue for your business. The benefits of WPO are many; let us discuss the key ones.

- **Improved revenue**: Fast sites translate to more revenue. You can *Google* the web and see all sorts of data around how performance impacts revenue. How much you lose by having a slow site can be determined by correlation between bounce rates and how those bounce rates impacted the customer journey or usage of the product. But the hardest part to correlate is brand impact without larger datasets over time to determine how it impacts future business and contracts. Everything you consider around performance should be able to be tied back to revenue and this is what you should shoot for to justify why performance matters. If you are brave, try introducing increments of 500ms of latency into key flows in your production system at peak load and understand how latency impacts bounce rates. Then correlate that to the revenue stream. You cannot make good decisions unless you know that you are going in the right direction.
- **Brand reputation**: Without performance beyond your competitors, you risk not only losing potential traffic but also hurting your brand's reputation and impacting revenue streams. Users tend to associate slow and poor usability websites with poor service and unworthy products to consider. Not to mention, the renowned search engine algorithms which favor sites with faster load times could sneak your

website lower in the pecking order. Not investing in WPO could leave you playing catch-up in the competitive world of online business.

- **Website speed above competitors**: A crucial element of web optimization is improving your website's load time. The perception of speed means the product looks solid. Internet users are increasingly impatient and will leave a site that does not load within a few seconds. An optimized website can improve user experience and increase visitor retention.
- **Better user experience**: An optimized site provides an improved user experience. With a user-friendly layout, responsive design, and easy navigation, users can find the information they need without unnecessary hassle. This encourages return visits.
- **Better search engine ranking**: Search engines use software algorithm that prioritizes optimized websites over others when displaying search results. A website optimized for performance and user experience ranks higher, which means increased visibility and attracting more traffic.
- **Increased conversion rates**: A user-friendly, fast and efficient website not only attracts visitors but also fosters engagement and promotes conversions. WPO, as a result, is instrumental in boosting returns on your digital marketing investments.
- **Security resistance**: Optimizing platforms for speed and performance assures easy detection when something goes wrong and leveraging tools for edge caching many times also provide the benefit of preventing **denial of service** (**DoS**) attacks.

Web performance optimization

Let us go deeper into how to create fast websites and platforms and WPO. Remember that the speed and efficiency of a website could be the deciding factors between the success and failure of a business. WPO, is the practice of making websites or web applications more efficient. By enhancing an IT platform's infrastructure and architecture to support the needed efficiencies, leveraging the proper timeouts, connection pooling, keepalives, balancing algorithms, caching, etc., then optimizing code to assure good performance patterns are followed, perfecting UI/UX designs, and adjusting the server's performance; developers or performance engineers can significantly minimize the loading time and increase the capacity of these platforms. The result is a great customer experience!

TTL caching optimization

In basic terms, TTL stands for **time to live**. When you are designing a multi-tier distributed architecture, TTL becomes very important. It is a crucial component of various networking and caching strategies. When we discuss the benefits around TTL caching, we are talking about the importance of defining a lifespan to cached data, determining how long it should stay in the cache before being refreshed or discarded. In many cases, this may mean

months and in others, minutes and it depends on how frequently the data is updated and how important the timeliness is for said data.

One of the most notable benefits of TTL caching is its ability to enhance content delivery speed to a web browser. By storing frequently requested data closer to the end user and refreshing it only, when necessary, the application or website can provide quicker access to information and not need to make a large number of slow requests across the network to a web server, for example. This results in a smoother, more efficient user experience, as the need to retrieve fresh data for each request on each page of the web application is minimized.

TTL caching is especially beneficial for data that changes at regular intervals. For instance, consider stock market updates or observability metrics. This type of information needs to be accurate but not instantaneously refreshed every millisecond. Setting an appropriate TTL allows you to strike a balance between keeping data fresh and reducing the frequency of cache updates, which can otherwise put load on the servers impacting the platforms capacity while degrading the user experience.

Using TTL caching simplifies the caching strategy. Since it is so easy to understand and implement, it does not require anyone to be a rocket scientist to understand what object is cached and what is not. Understanding how various caches interact helps in setting intelligent default TTLs. By fine-tuning the TTL settings, you can manage how quickly users receive fresh data and how frequently the cache updates. A shorter TTL might ensure more current data but can lead to increased server load due to frequent updates. Conversely, a longer TTL reduces the server load by limiting updates but runs the risk of serving outdated data. It can be amazing to see how much just a 5 minute delay can have on improving performance for your customers and capacity for the platform.

TTL also plays a vital role in managing how long content remains cached on edge servers. When users access content that falls within the TTL period, they experience much quicker delivery speeds compared to retrieving the data directly from the origin server. This not only improves performance but also enhances the scalability of your application, as the origin server is under less pressure from repetitive data requests.

Ultimately, understanding and utilizing TTL caching can profoundly impact the efficiency and responsiveness of software applications. By optimizing TTL settings, we can assure that users receive data quicky and have a great user experience.

Optimization tools

Understanding and harnessing the potential of WPO becomes exceedingly critical to the success of every IT platform. But where do we get started when it comes to optimizing performance? Tools like Lighthouse reports or Performance Insights browser features, readily available under the development tools menu option in the Chrome browser, do provide a lot of good starting points as far as coding for simple best practices. These should be leveraged early in the development process.

observations that we can make with flame graphs and metrics collections through profilers, but that is not always the best place to start when it comes to optimization. It is like going into the details before taking a high-level view of where the problem exists first. What tells us the most when it comes to the web at a higher level before we do deeper? Where do we start?

Remember that performance really is just an aggregate or the sum of the parts if you will. Sure, some of those parts run in parallel while others in a serial fashion but they all are finite pieces of the aggregate performance measurement around the page load times. To represent this, we always start with the waterfall as a first step. When it comes to analyzing web performance, a waterfall is the best visual model to review that showcases the detailed performance dynamics around how a webpage loads and the elements and resources referenced in its source code. It will show queuing times for threads, download times, DNS lookup times, etcetera. It first starts from the initial request and records what happens for the entire load process at a granular level until the page finishes loading. This waterfall view exists in many vendor observability products and also within many browsers and is invaluable in the art of WPO. It is shown as a sequence or timeline of events, stacked linearly like a flowing waterfall. For bottlenecks, you want to look for where you first start to see queuing for threads to be available for rendering. This is your first bottleneck when queue times start contributing to too much latency.

Waterfall optimization

Many browsers put the waterfall view on a network monitoring tab embedded in the browser itself. When loading the page, look at the bars being showcased around each request. Each horizontal bar on the waterfall denotes different URLs or resources involved in webpage loading, like HTML, CSS, images, JavaScript, and much more. The waterfall can allow software developers and performance engineers to pinpoint web performance bottlenecks and inefficiencies very quickly, facilitating enhanced WPO.

When it comes to WPO, this is one of your most valuable tools. If you have been in the web developing world for a while, you might be already familiar with it and have seen the advantages. People familiar with WPO for many years will remember first seeing it leveraged in popular observability solutions on the web like:

- GTmetrix
- WebpageTest
- Sitespeed.io

Waterfalls work to show you what is blocking your page load and how they do this is fascinating. Remember that each component of your webpage is like a section of a waterfall representation of your page loading in this page load view. When you watch it loading, some parts will load faster, cascading down quickly, while others might take a little longer. The dynamics can also be altered depending on the location of the customer as well. When viewing **Real User Monitoring** (**RUM**

you will see different waterfall results from someone in *India* versus the *United States* as a simple example. This means the rendering dynamics and the customer experience can be completely different. A large image that is 500Kbytes in size may load faster in the *United States* and block the browser threads for less time than in other countries because of the closeness to the origin server. The time when the page becomes truly interactive is what can be impacted.

Geo-location optimization

A good way to understand the true experience of your customers is to run a load test at both peak load and at nominal load for the platform locally while requests come in from different locations. Which locations to use? You first need to understand where your customers are coming from. Using analytics tools is one way to do this. Once you understand where the traffic is coming from, which browsers, which devices and the percentage of traffic from each location as far as workload, you can determine how best to test and analyze a website.

Some testing solutions will allow you to run tests from various geo-locations around the world. We have not found this to be beneficial except for determining the dynamics around **content delivery networks** (**CDN**) edge caches under load and standard deviation around the testing infrastructure being used from cloud containers and different virtual machine horsepower configurations. We will talk more about CDNs later in this chapter as far as usage. But from a testing perspective and when following a good performance practice, always run the test as close to the origin infrastructure as possible and then run a few requests remotely in each region to understand the standard deviation around response times for what is being perceived as rendering latency. This last part can be done manually.

You want to see what you can optimize at the origin using this approach. Then, try testing the path with the CDN edge by caching the website to see how it impacts the dynamics. Run a test and confirm it and look at CDN optimization or scaling up those servers. The problem with running load with load generators across geo-locations is that you are running on an **internet service provider** (**ISP**) server or cloud servers that you do not control, which may be contributing to the latency itself and not representing the same dynamics as a user's ISP. Make sure that you know the latencies being seen at origin always first. Tune it there for the best experience and then look at the other dynamics and tune back through the layers.

Cross browser/Device testing

Testing with different devices and browsers that your users leverage will allow you to see the rendering dynamics which differ not only by location, but by browser and device thread dynamics. Running observability tools that visually showcase what customers experience in a session recording will help you gain insights into how the different waterfall dynamics impact rendering and usability for a website visually beyond the

and provides actionable insight into what is slowing your website down. Seeing it visually with session recordings can really showcase the slow rendering experience that the user may be experiencing with the platform.

In an ideal scenario, you want all artifacts and components on a website to load simultaneously, right? Unfortunately, this is not realistic and any delay in one element could lead to further delays in the subsequent ones, affecting the entire page load time. In fact, if you get creative and increase browser threads for yourself to something like 100 to represent every element on your page to run components in parallel, you will look like spam to many infrastructures and generally get blocked. No infrastructure likes to see 100 requests coming in all at once from a single user. They will think they are under a DOS attack.

Understanding the waterfall model can significantly help optimize your website's performance. Several factors on a web page can slow down the waterfall, creating a domino effect that impacts overall performance. This typically includes large files that require more time to load, such as high-resolution images or videos and even large frameworks that have not been properly split up or optimized. Improperly optimized images or bulky code will also contribute to delays. Moreover, inefficient JavaScript and CSS, as well as excessive HTTP requests, can significantly impact your webpage's page load and rendering speed. Other factors, include server response time, the physical distance between your user's location and your server, and a poorly structured website that does not prioritize loading of above-the-fold content (page view without scrolling).

Beyond the waterfall, you can also drill down into JavaScript profilers, Java runtime profilers or the language that you are using for your IT platform development on the backend/frontend. Beyond waterfall view of your page load, you want to focus on specific metrics when it comes to tuning web performance. These are metrics that you will want to have SLOs represent and what you will augment and optimize the IT platform to improve. Let us walk through these and explain each one.

Core Web Vitals and best practices for web performance

Google came up with metrics that it considers the most important aspects for fast page load time. These metrics consider content loading speed, responsiveness and content stability. The three Core Web Vitals defined by *Google* are **Largest Contentful Paint (LCP**), **First Input Delay (FID**), and **Cumulative Layout Shift (CLS**). LCP measures the loading performance of your website, ideally this should always occur within 2.5 seconds. If it does not, your website needs optimization. The second key metric is FID, it quantifies interactivity, aiming to be less than 100 milliseconds. It measures the time from when a user first interacts with your site (i.e., when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to start processing that interaction. The final metric is CLS. This deals with visual stability, and it should be less than 0.1. It quantifies the amount of unexpected layout shifts of visual page content. Before Core Web Vitals, everyone used **Time to Interactive** (**TTI**) but venders like

Akamai may have a different TTI definition that *Google* which became problematic. LCP is what to set SLOs to when verifying thresholds for quality gating and is where to start tuning to improve performance first.

The following is the short list for the most important processes that should be put in place to always assure the website is fast and to improve LCP?

- Keep a small page weight
- Minimize HTTP requests across the network
- Combine CSS or JavaScript files where possible to reduce trips across the network
- Combine multiple small images into single files such as sprits for one request
- Minify CSS and JavaScript files
- Defer JavaScript or run async to prevent render blocking
- Compress all content including JSON going across the network larger than 5Kbytes
- Always use the lossy optimization algorithm for image optimization where duplicate pixels are removed versus lossless approaches that just remove EXIF meta data for all images, not just large ones to a threshold of 80%. This is a point where the naked eye cannot differentiate differences in images. We talk about this one more later in this chapter
- Cache static content in the browser for long durations (30-90 days)
- Do not use ETags, leverage fingerprinting techniques (change hash or version naming)
- Profile HTTP1.1/HTTP2/Http3 and implement fastest solution for rendering
- Use lightweight web frameworks like React/Vue with Virtual DOMs
- Implement fast services or use async eventing to prevent render blocking
- Use optimized runtime environments
- Leverage a CDN
- Denormalize the data storage or use NoSQL and implement coarse grained retrieval for services, index and have results waiting in memory
- Heavily leverage browser web service workers and HTML5 storage for pervasive computing
- Eliminate redirects
- Cache **HTTP Strict Transport Security** (**HSTS**) in the browser

Many times, websites need to be accessed around the world. If the origin server for the website is in the *Midwest* of the *United States* and the website needs to be accessed by someone in *Japan*, there would inherently be some delay.

already about how the waterfall view, when analyzing websites, can be impacted by location. But how do we account for this and improve it? What can we do about it? Beyond streamlining the loading of artifacts using the techniques described above, we use something called a CDN. CDNs do the following:

- Improve website loading time in different regions of the world
- Boost website security
- Support high traffic volumes
- Reduce bandwidth consumption
- Improve performance which improves SEO Ranking

Content delivery networks

Let us break it down for you to understand how a CDN, or content delivery network, operates. Essentially, it works by hosting your website's content across various servers located around the world. Whenever a user pays a visit to your site, the CDN fetches and delivers the requested content from the server closest to the user. The goal here is to minimize the distance information needs to travel from the server to the user, which, in turn, often reduces website latency and improves loading times. In this way, a CDN optimizes web performance by ensuring that users across the globe can access your content as quickly as possible, no matter where they are located.

Generally, the CDN will have cached content and static content and periodically go back to the origin server for updated information. It may do this once per day or once per hour, even less depending on how often content changes. Many advanced CDNs these days also support dynamic content schemes for short durations or have ways to know when the data is not up to date. They have abilities to remotely trigger flushing of caches around dirty data. These solutions like *Akamai* have a host of features. But, the most important features are caching, optimization and are security related. Let me warn you, some CDNs will advertise that they automatically optimize your images. This has been hit or miss with the CDN trying to please all customers and not reducing or optimizing images as far as they could be optimized.

Always optimize images regardless of size

Always optimize all images at the origin server. This reduces the payload across the network meaning faster updates to CDNs, more scale, and also assures that you have maximum compression with image being lossy optimized.

What do we mean by lossy? There are two ways to optimize images. One is lossless image optimization which means removing the EXIF data from the image which is the metadata around the picture. This will reduce the file size by ten percent which is a nice benefit to load times. In the perfect world, a ten percent reduction in file sizes reduces load time by a

The second way to optimize images is called **lossy optimization**. This means that the image will be analyzed pixel by pixel around colors and clarity and where redundancy exists, information is removed. Depending on how much information is removed, the file size will significantly reduce. All images should be lossy optimized if you want to see great performance and this optimization should not limited to just the large images. You can generally get by by removing twenty percent of the image content and visually not be able to detect the difference on a computer monitor with the advanced tooling today.

Web editing tools like *Adobe* and others are not good at web image optimization, in our opinion; however, they are extremely good at creating and editing images. Leverage tools like ImageOptim (**https://imageoptim.com/mac**) or something with similar optimization levels for your IT platform. You want to balance image quality, display size, and file size. You want the highest quality in the viewport size your customers use with the smallest file size, which means the lowest bandwidth consumption across the network. Remember that you want to optimize all images in this way, not just the large ones. A 10Kbyte file can reduce down to 2Kbytes in size which means the data is transferred across the network faster down to the user's browser and that the threads in the browser will be held onto for less time and not block other content meaning a faster rendering time. The image formats that you have on your website are generally:

- **JPEG**: An image format best used for photos where color detail is important.
- **PNG**: A great choice for computer-generated images or photos requiring transparency.
- **GIF:** Ideal for simple animations.
- **WebP**: a modern image format that provides superior lossless and lossy compression for images on the web.
- **Jpeg XL**: A modern image format optimized for web environments. JPEG XL generally has better compression than WebP, JPEG, PNG, and GIF.
- **AVIF**: a modern image format based on AV1 video format.

To see where an image format can be used or capabilities in different browsers around HTML 5, leverage **https://caniuse.com/**.

It is important to understand that using a CDN does not mean that you do not need to cache content in the browser. You want to always cache content in the browser. As a practice, what you want to do is minimize requests to every layer to build scale and have a faster response for your customer. This means leveraging HTML5 features, service workers, non ETag approaches to content updates, unless that content is updated daily or more frequently and remembering that no single request should go across the network that does not need to be made.

The time it takes for packets to flow across the network varies by how many hops are taking place and the bandwidth, but it can be larger than people think. Look at network

times in the Chrome developer tools suite for your website and you will set what we mean. This time is additive and if you traceroute the packets across a network though the various switches and servers, you will see that there is a lot of standard deviation in response times around packets. It may take 20ms for one request and that same request may take 50ms an hour later. You may even encounter packet re-transmits taking latencies out as high as 500ms or even 1 second. When it comes down to individual granular network routes, if using Linux, we can leverage a simple tool like **mtr** to do the performance analysis we need.

Multi-layer caching

Remember, beyond addressing network issues to achieve great web performance and give as much bandwidth as possible, you want to cache at the browser, cache at the CDN, cache at the reverse proxy and cache in the pod, virtual machine, operating system and in the JVM runtime environment. Each layer should have some level of caching. Never make a call across the network between each layer that does not need to be made. For example, if you need to cache the 50 states of the *United States* in your application which does not change very often, should this be a database call through a service in real-time? The answer should always be no, it should always be cached data. It is fine to poll on a given frequency or trigger a pub/sub type of even to update the cache for less static data as well, but it is cached data. Is it fine to replicate cached data across pods, notes, machines, etc.? The answer is yes! Small amounts of frequently requested static data should be cached without going across the network local to the application running. This can be true for dynamic data if an event can be triggered to flush/refresh the cache from a database or other service with direct or pub/sub events being consumed. Remember to always have a mechanism to refresh the cache without restarting the service.

When it comes to packet loss and requests across the wire, bandwidth impacts, etc., it is a good idea to measure network times throughout the day to understand average packet latency or monitor at a higher level to determine the time of day when latency is worse and keep retransmits less than 2%. You will find that usage drives bandwidth changes and understanding that correlation is important. If you have a network with speed of 200 megabits per second upload and 400 megabits per second download at 8:00 am in the morning, at noon, it may be completely different.

Resource consumption impacts network bandwidth and page load times. This is why we want to always reduce the amount of information sent across the network and leverage caching. Always consider your end-to-end journey from the browser, through the CDN, to the origin servers to the application. You will see the waterfall for the application change based on regional requests. Of course, it may take longer for one request versus another that blocks a resource and impacts rendering times. In other words, rendering times in *Japan* may not be the same as in the *United States*.

How else can we assure a fast platform beyond improving rendering times using a waterfall analysis? Is there anything we can do speed up the experience? The best approach to

improve performance and IT platform capacity is to look closely at the top ten journeys in a web application. If you know the most used journeys in your application, you not only can focus on what drives capacity, but you and predict user behavior. If you can predict user behavior, you can have content ready and waiting and thus give the impression of and achieve extremely fast performance. At the end of the day, what is important is perception of speed and quality. This is the same reason we put a priority on above-the-fold content. It gives the perception of more speed even though you may be lazy-loading objects below the fold at a slower rate.

Performing content retrievals ahead of time

What can we do to improve quality once we know the top ten journeys and can predict the user behavior? Let us delve into how the attributes preload, preconnect, prefetch, and DNS-prefetch and see how these powerful attributes on web pages contribute to enhancing web performance. Starting with **preload**, this as an effective tool often utilized to load key resources that are needed for the current navigation. It gives you the power to inform the browser about critical resources it needs to load for a particular page ahead of time. The benefit you get from this feature is that it allows your website to load faster, resulting in a remarkable user experience. What about the slowest part of every request to a third-party dependency or service on another domain? This is where **preconnect** comes in. This capability offers an extraordinary advantage for your website. Preconnect allows the browser to connect and establish early connections to a third-party dependency or downstream service, before an HTTP request is needed to actually sent a request to the server. This includes DNS lookups, TCP handshakes, and optional TLS negotiations. With this in place, your site essentially cuts out the real time overhead, eliminating potential delays and providing a faster, smoother experience for your site's visitors.

Moving on to the **prefetch** attribute**,** we start to see some real capability and power to push performance; this technique does not get nearly enough press in our view. Prefetch is what can be used to load resources and content that will be needed for future navigations or user interactions. This implies that when the user moves to the next page, the resources are already in the cache, and the page loads instantly. This clever proactive loading of artifacts assures a fast and seamless browsing experience for your users, putting your website ahead of your competition.

Finally, we arrive at **DNS-prefetch**, another incredible tool for optimizing your website performance. DNS-prefetch implementations will proactively perform domain name lookups in anticipation of what a user might follow. DNS can be slow and if DNS caching is not in place, it can be very slow. Doing DNS lookups ahead of time, can really boost some performance without a lot of content coming down across the network. This takes the edge off any potential latency around DNS, leading to a more responsive and engaging user experience. Not only does it save your users time, but it can also make your site feel more fluid and responsive, a factor that can significantly impact their overall experience.

In essence, leveraging these four web attributes; preload, preconnect, prefetch, and DNS-prefetch, can dramatically enhance your website's performance and rendering times, ultimately leading to increased user satisfaction and more frequent visits. It is no exaggeration to say they are some of the best optimization tools in your arsenal and something that all developers should be leveraging when creating web frontends. Remember to always leverage these when building software. Once you understand customer patterns when using a web application, you can really make a difference.

Performance design patterns

Another optimization and mistake we commonly see for web applications is going to a landing page for a web product where a hundred queries and interactions or events occur during the page load. Nothing and we mean nothing is more important than that first page being fast. The first page is where someone using your site for the first time is going to have an impression of the product and your brand. To achieve great perceptions, you want to keep it lean and loading fast!

Design your application in a way that the landing page can come up with minimal backend interactions. This is not the page to have 100 backend requests that are calling ten different micro-services to retrieve and correlate data with the backend datastore. It is much better to have pre-cached content that drives your first page if it must be dynamic or have static content that then interacts with the backend services when the user clicks an option. Always take advantage of any browser idle time after page load to start loading the most use path or paths behind the scene if there are natural flows that users follow. Then when the user clicks those paths the impression is optimal performance.

One good approach for websites that require users that login to a site first is to have a user profile with associated data cached ahead of time before the user first starts up the site. If the user has not logged in yet, remember to always have default common content cached. If you have one hundred thousand customers and you do not feel that is practical, load content into a cache or memory for customers that have logged into the site or used the site within the past 90 days, including new registrations. This way new customers on the platform are assured to have a great experience and customers actively engaged with the platform will always have a great experience. Only people that rarely using the platform, if at all, would have a longer page load going after non-cached content. These types of users there is no benefit in caching or incurring the expense to cache so much data.

These types of forward-thinking caching strategies driven by user behavior can make a huge difference to perceived performance. What to do if you have slow logins? Take the same approach as mentioned above. Keep the data in memory based on usage of the customer and have those users be able to login in less than a second where other users that rarely use the platform may encounter a 10 second login time. When it comes to caching very large datasets, you either breakdown caching by taxonomies and relationships as far as the size of the cache, along with levels of caching, or you cache everything that is needed for the first page (and more) based on behavioral usage patterns. You limit caches by behavior versus a taxonomy of data relationships.

Security can be painful for performance. SSL caching is definitely a requirement across architectural layers to achieve good response times. Leverage it heavily. Where security encryption or hashing comes into for **personal identifiable information** (**PII**) or e-commerce data on a website for **Payment Card Industry** (**PCI**), remember to encrypt/decrypt in a very coarse-grained way when it comes to large data streams. Encryption/Decryption is one of the most CPU intensive slow operations, you want to do it in a coarse-grained fashion and do a lot of data at once versus a single field where you need to do it ten times. In other words, you want to use a key with a single request to decrypt or encrypt in one operation for a network payload and not encrypt/decrypt data in a granular way. Put a lot of CPU power to encryption/decryption operations to help with performance. When using a database that supports multiple tenants and encryption, you want to leverage the database driver that supports seamless encryption/decryption. Never try to start encrypting and decrypting data outside of the database yourself for queries and updates or you will quickly discover how slow database can be. Always leverage native drivers written in C language optimized for performance when it comes to seamless encryption/decryption with queries or updates to the database. If for some reason you have no choice, then encrypt or decrypt an entire blob/clob of data containing multiple fields versus single fields. You get the idea, minimal encryption/decryption operations should be executed against large datasets or payloads for the best performance.

When components startup or databases startup, precaching should be happening. This is referred to as **heating the cache**. The cache should be flushed, updated going forward by a trigger event when something changes. Polling for changed content such as from a database is not ideal, but that approach is another option. Triggering by events though, assure everything is happening in real time which provides you a resilient application with valid data. For example, of the data goes dirty and you have 10 nodes, you publish to an event bus and those 10 nodes update or dump their data depending on the situation. Remember for smaller caches, you want to cache at the application layer and use events versus a centralized enterprise cache where you encounter latency across the network to retrieve the content in real-time. You can also take direct event approaches beyond pub/sub types of busses, but that can lend itself to more complexity.

Remember to consider failover and disaster recovery exercises when optimizing platforms. If you have balancers or components failing over to different regions, this impacts the rendering workflow and performance/scale of the application. You need to test for these scenarios and make sure that thread pools, connection pools are set appropriately to handle the new dynamics. If you go from one region to another, you may find that transactions will slow down and impact the customer experience. Remember that you want to test LCP and make sure that the same experience exists for the customer in this situation. This goes for backend services as well which ultimately impact LCP and rendering dynamics.

Performance is not only determined by the resources being brought down from a web server to a user's browser, it also depends on the protocol being used. Understanding HTTP versions will give you a much broader perspective on the behind-the-scenes mechanisms that run the internet. Let us start with HTTP 1.0**.**

protocol for transmitting hypertext requests from a client to a server. The main limitation was its one request per connection policy, meaning only one file could be loaded at a time. This led to slower loading times and less efficient communication. When HTTP/1.1 came out, it was an upgrade that improved upon this limitation allowing for multiple requests to be made on a single connection. However, the requests were still processed one right after the other, maintaining a linear loading sequence known as head-of-line blocking. It meant substantial waiting times if a certain request was to delay, underlining a need for further improvements.

HTTP protocols that impact performance

In 2015, the HTTP**/2** protocol was introduced and showcased the concept of multifaceted/multiplexed communication that revolutionized web performance. This included the ability to send multiple requests and receive multiple responses simultaneously on a single connection. This innovative approach to communication across the network for page loading, significantly improved page render times and overall speed for many sites. It has been heavily adopted today throughout the industry. HTTP/2 also has the ability to do a server push, a feature that preemptively sends data to the client's cache to optimize loading speed further still. Improvements have continued over the years with the introduction of HTTP/3[1], which incorporated the QUIC protocol. QUIC ensures that multiple streams are independent, so packet loss in one will not affect others, resulting in a smoother, more reliable online experience.

Understanding rendering workflow dynamics

When we talk about performance, sometimes it may seem pretty confusing as to what web latencies are related to the visual experience and what is related to backend processing for a web page. There is a lot of information on the web about rendering and charts with graphs showcasing the interactions with some fancy automation, but let us go as simple as we can to understand the dynamics. Sometimes stepping back and looking closer at things in a simple way can drive more clarity than a graph representing all of the rendering dependencies. Let us start digging much deeper to fully understand how web page rendering works by starting with the basics. We will start by looking at a very basic HTML page to see how it impacts performance. This is one that essentially just provides a simple basic HTML body with a title and one image. We want to create a very typical, not over the top type of welcome page to showcase, as a good example around what is happening around rendering:

1. https://caniuse.com/?search=http3

```
<html>
 <head>
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <title>Let's look at the MasterCard Critical Path: No Style</title>
 </head>
 <body>
  <p>Happy Friday from <span>web performance</span> Vince!</p>
  <div><img src="awesome-mastercard-photo.jpg"></div>
 </body>
</html>
```

This page will scream with performance less than 100 milliseconds in most cases. Then let us look at the rendering flow and try to better understand what is going on:

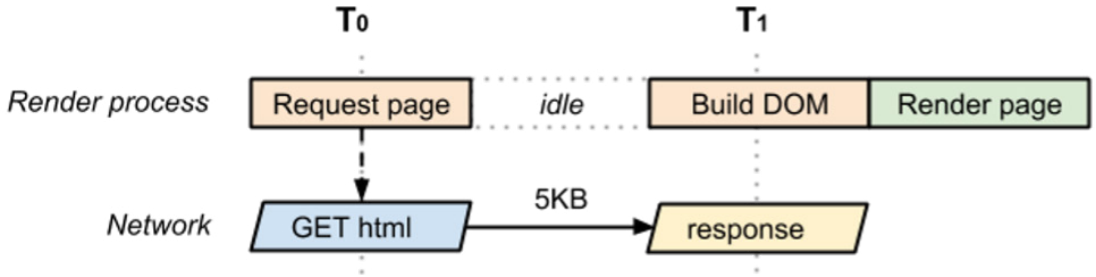


Figure 6.1: *Initiating a browser request*

In this basic HTML example, the time between *To* and *T1* captures the network and server processing times. But the question and contributors to latency quickly becomes based on how many round trips do we need to fetch an entire HTML document with the information that we have embedded in the page using a web server. What we see is that such a basic HTML page will visually render extremely quickly. You will no longer than hit a button on your keyboard, after typing in the URL, and the results will be there staring you in the face. It will render like the *Google home page* which is very well optimized for performance. So, with only HTML elements and no external JavaScript or CSS, things render pretty quickly. Why? Primarily because no external trips across the network is occurring. No sending and receiving of network packets beyond that initial page is happening. Now let us add some of that pretty CSS everyone likes and reference an external stylesheet and see what this does to rendering (stylesheets are used to get some great visual appear to fields/content):

```
<html>
 <head>
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <link href="style.css" rel="stylesheet">
 </head>
 <body>
  <p>Happy Friday from <span>web performance</span> Vince!</p>
```

```
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

The following figure shows building the response:

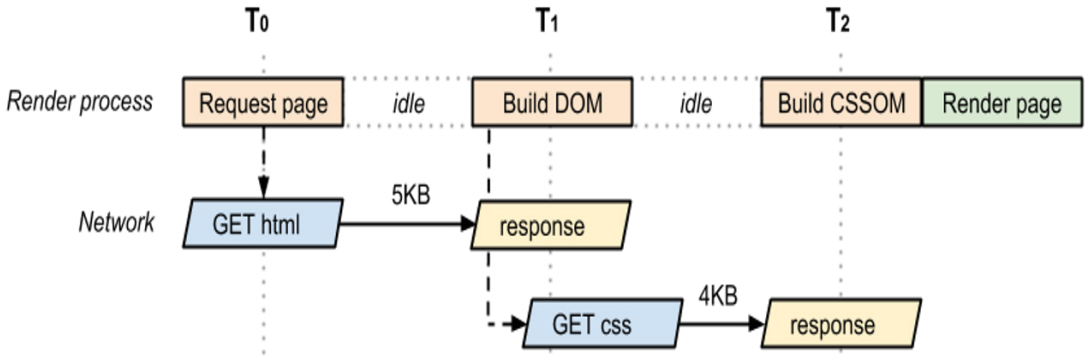


Figure 6.2: *Building the response*

Look at the flow differences. Now in order to build the response and render the page we have more steps. Before we can render anything, we need to reach out and get this CSS file across the network to render it. This is not the case for embedded CSS statements in HTML. Almost every CSS file that exists tends to be a render blocking resource which means the browser has to download them before any other content can be rendered on the screen. It quickly becomes a blocker and the more import or external reference statements a CSS file has, the more blocking that occurs through more trips across the network in many instances and bottlenecks to rendering. The size of that file can greatly affect rendering time due to the time to take the file across the network. The smaller the file, the better for performance and faster rendering times. Make sure that the CSS file is both minified and compressed to improve response times and keep file sizes as small as possible. Combine CSS when you can to reduce trips across the network. This page will incur more roundtrips across the network before the page can be displayed if there are dependencies and files are not combined.

The CSS download and rendering may also slow down due to network packet retransmits as well so make sure that you monitor any bottlenecks across servers through observability. You want to keep retransmits small. In a nutshell, this page is much slower now because it cannot render until it grabs all of that information from across the network. Next let us add some JavaScript.

```
<html>
 <head>
  <meta name="viewport" content="width=device-width,initial-scale=1">
  <link href="style.css" rel="stylesheet">
 </head>
```

```
<body>
 <p>Happy Friday from <span>web performance</span> Vince!</p>
 <div><img src="awesome-photo.jpg"></div>
 <script src="app.js" async></script>
</body>
</html>
```

The following figure shows how scripts impact rendering:

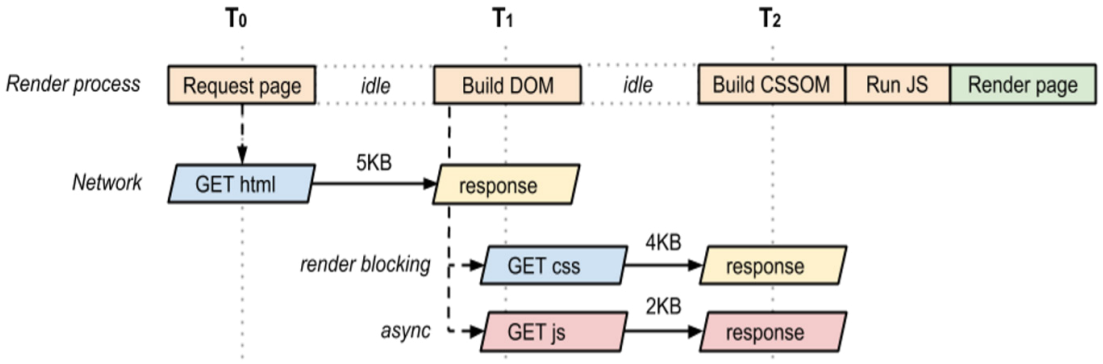


***Figure 6.3**: How scripts impact rendering*

When we include HTML, CSS and now JavaScript, look at the number of paths and routine trips across the network that need to occur. Wow! So, adding that JavaScript file means even more round trips beyond what CSS needed. If you want a page to render very quickly without any blocking from scripts, you want to consider some things. For CSS information that is very small, embed it in the page. Yes, embed it in the page so that request across the wire does not need to happen. If you have a lot of CSS, combine it all into one file and minify and GZIP (or other like Brotli) compress it so only one request across the wire needs to happen (compression can happen in JBoss, Apache, Nginx or on another server). But what can we do about that JavaScript file throwing a wrench in the performance works by blocking rendering? Easy, use defer and async attributes on the script so that it runs in parallel with the browser rendering. If you cannot run async (and you always can for scripts like analytics, etc.) then use a JavaScript loader that will map the needed execution dependencies for optimization. A JavaScript loader will allow you to specify your dependencies and still parallelize as much as you can to keep from blocking rendering. When it comes to performance asynchronous processing is your friend.

To build really fast pages, always start with the simple HTML basics for fields and minimize overhead, images, etc. When you absolutely need CSS, then use it in small snippets to add visual appeal. The same goes with JavaScript embed small snippets on the page vs. going across the wire. When you use images always specify the height and width attributes on the page so the browser does not need to figure it out to render.

What about large frameworks that slow down page loading? Perhaps you like to use large frameworks in your development that have high levels of abstraction to speed up the development cycle. You inadvertently end up piling on that bulky JavaScript, CSS, UI libraries; adding to the page weight, just in case you may need it.

UI frameworks are great and help us a great deal, especially when it comes to application operation across browsers and viewport sizes. They provide great benefits especially with **Responsive Web Design** (**RWD**) today where different layouts exist for different devices and only one code base. But sometimes certain UI frameworks can be a curse as well. Many organizations choose frameworks based on their great feature set.

Organizations almost always forget about the most important elements when evaluating development frameworks for their IT products. They forget to consider performance, security or unnecessary bloat. We always recommend that do a **proof of concept** (**POC**) with each framework you are considering and benchmark it. Yes, measure the performance and the performance impacts around load times into the browser. When it comes to framework creators, admittedly, it is hard to add a million features and still have great performance no matter how we try to split up files. So there needs to be a balance for us to meet our goals. To do that, we want to always use lighter weight web frameworks like React or Vue or one of the other streamlined frameworks, etc. and even then, eliminate any code you are not leveraging from a CSS/JS perspective. Even if you are using JavaScript **Model View Controller** (**MVC**) types of frameworks, consider performance and streamline those extra features that are not needed.

If you must use larger frameworks like AngularJS where organizations have not yet moved to the performance-oriented frameworks, remember to remove debug code in production, remove JavaScript not being leveraged, leverage the optimization features of the AngularJS framework for prod deployments. Make sure minification is always there and files downloaded to the browser are as small as possible. Also leverage great tools like Webpack to code split and bundle assets. There is a plugin available to this framework called the web-bundle-analyzer that can really make a difference. This framework has some great optimization features built into it as well beyond all of the many great plugins around optimization and compression that speed up websites (**https://webpack.js.org/configuration/optimization/**).

Be conscience of how many round trips you may be making due to the size of the files. The smaller the file the less trips/packets across the wire. So, if you have that million-line JavaScript and CSS library because you chose a bloated framework, consider going through and eliminating any source code in that framework that is not leveraged even if that means having your own company version. In most organizations, 90% of most features are never used. Always choose a light framework for great performance that gives you what you need the most along with great performance.

If you are lucky enough to be going through the process of choosing a new framework, make performance objectives with security features the highest consideration on the priority list

bulky and many organizations leverage only about 5-10% of their functionality. Instead, choose frameworks based on performance and core needed capabilities versus that fatter slow one with the great feature set. If a framework has no performance benchmark stats, you do not have time for a POC, or you cannot find a site to run existing performance metrics on it, my recommendation would be to completely stay away from it. Bigger web frameworks are not always better. Look at performance, security and market adoption and you will never go wrong as an organization.

IFrame page load impacts

IFrames, also known as **inline frames**, are a crucial component of web coding, serving as portals that link different websites and, essentially, make the internet what it is today. However, mastering the performance optimization of IFrames can be tricky. This article will guide you through the journey! First how can we make IFrames load faster? Here are a couple of tricks.

- **Lazy loading**: What we care about when it comes to initial page loading is what is above-the-fold, or when is displayed initially without scrolling. Sometimes, IFrames can be a drain on the page's resources, especially when several IFrames are being loaded at once. This is where lazy loading comes into play. With lazy loading, IFrames only load when they become visible on the screen, hence saving resources. Modern browsers now support native lazy loading with the `loading=lazy` attribute.

- **Harnessing the power of cache**: Storing the static content of an IFrame in the browser's cache can up its load time significantly. This technique serves previously saved content to your users, eliminating the need for it to be fetched and loaded each time they visit. It is a smart way of reducing the workload of your IFrame, making it run more efficiently. It is usually used for static content, i.e., information that does not change frequently. For example, images, CSS, and JavaScript files. It increases the speed and performance of your IFrame by reducing the load time of these static elements, especially when users revisit your webpage.

- **Configure set expiry headers**: Set expiry headers tell the browser whether they should request a specific file from the server or fetch it from the browser's cache. Configuring these headers correctly allows you to tell the browser which files to cache and for how long.

- **Minimize HTTP requests**: Every file, be it an image, a style sheet or a script, equates to one HTTP request. The more requests, the longer it takes for the entire page to load. Such unnecessary requests can be minimized by combining files, like merging all scripts into one script, and similarly, combining all CSS into one style sheet.

The downsides to IFrame

Is there any disadvantage to using an IFrame? The most prevalent reason you will find for avoiding IFrames is due to the impact to SEO metrics. Why? Because if you embed an entire page in an IFrame, the search engine robot crawler does not always interpret the website in the same way as someone not using an IFrame. The content in the IFrame is many times not picked up by the search engine. So, if you have content that says has a product or term you want to be picked up in a search engine, it may not be. Due to the IFrame, the search engine does not display the page like it would if you had the word credit card in the main page.

Unless you introduce extra JavaScript, the user will not be able to obtain a link directly to content within the IFrame. This includes links someone might email a friend, bookmarks, or pages that come up in a search engine result. Imagine your user goes to your page, and some content is in an IFrame. They click a link in the IFramed content. This content will stay within the IFrame as expected. Now your users find some content they have navigated to within the IFrame and would like to send it to a friend. But copying the URL in their browser will not work, that would just send the friend to the initial page, with the original content displayed in the IFrame.

IFrames are sometimes used as a development shortcut to development processes for heavier large interactive pieces of content already created on another site, Though, anytime you call out to a separate domain on another server, you always run the risk of **Time to First Byte** (**TTFB**) latencies which may affect the webpage rendering times. The side effect of having multiple IFrames is having a terrible customer experience with render timing being difficult to control. TTFB is a key performance metric within WPO. It measures the amount of time it takes from when a user's browser makes an HTTP request to when it starts receiving the first byte of data from the web server. TTFB is a reflection of the speed and responsiveness of a website's server. The quicker the TTFB, the faster the web page can start loading, meaning a more optimum browsing experience for the user.

Remember that when using IFrames, they will block main page content from loading and slow down loading of web pages. What happens is a quirky experience and a slow load time. Though not ideal IFrames on HTML pages can sometimes be used as quick hits to bring in HTML content from other websites. They are often used to load third party content, ads and widgets for websites. The main reason to use the IFrame technique is that the IFrame content can load in parallel with the main page and it is simple to implement. You can get some content on a page from another domain very quickly. It is like a widget where you can place functionality or an entire site on a web page being developed and call it your own. Sounds like reuse at its best. However, at a very detailed level loading content in an IFrame does however have two downsides.

These issues are where the problems lie with respect to performance and user experience. On average web browsers use 2-6 connections in parallel for loading content from a website. This is why you see a stair step pattern on waterfall charts. When connections

are not available, the content is blocked on the page from loading. You want to always keep elements on a web page less than 100 for this same reason, thread blocking. The unfortunate reality is that only so many web artifacts can be processed in parallel on a web page. To optimize performance, it is important to leverage async and defer approaches to script loading or advocate using a good script loader if the script has dependencies.

The HTML **`<script>`** element allows you to define when the JavaScript code in your page should start executing. Parsing of the HTML code pauses while the script is executing. For slow servers and heavy scripts this means that displaying the webpage will be delayed. But you can use defer tags to delay script execution until the HTML parser has finished executing. If you do not care as much about timing, asynchronous script execution is the best of both worlds: HTML parsing may continue and the script will be executed as soon as it is ready. This is always the approach to take for things like analytics scripts.

When it comes to large latencies scripts and large images are almost always the culprits, for most websites. This is because they encounter a great deal of browser thread blocking when you have JavaScript or other page resources that prevent page content from coming down. Large images that have not been optimized to low file sizes tie up connections for a large period of time. This is such low hanging fruit and there are so many tools to optimize images (tinyPNG.com (**https://tinypng.com/**), JPeg-Optimizer.com (**http://jpeg-optimizer.com/**). Keeping those images small is critical to great response times.

In the context of IFrames, guess what happens when you have large images in that IFrame. It takes slowness to a whole new level! The onload event blocking is also a large issue. It is probably the larger of the two issues because it hurts performance the most. You really want the load event to fire as soon as possible, for optimal user experience and for search engine rankings.

Creating an IFrame

```
<iframe src="/path/to/file" frameborder="0" width="528" height="90"
scrolling="auto"> </iframe>
```

Using this basic normal IFrame technique will always result in the following behavior regardless of browser:

1. IFrame starts loading before onload of main page. It will use the same threads in the browser as the main page, meaning the main page will likely be blocked while what is in the IFrame loads.
2. Onload of the IFrame fires after all IFrame content is done loading
3. Onload of main page fires after the IFrames onload fires: The IFrame blocks the main page onload

To help with the uncontrollable rendering patterns, one approach is to defer the loading of the IFrame until the main page is loaded.

```
< script >
//doesn't block the load event
function createIFrame() {
    var i = document.createElement("iframe");
    i.src = "path/to/file";
    i.scrolling = "auto";
    i.frameborder = "0";
    i.width = "600px";
    i.height = "140px";
    document.getElementById("div-that-holds-the-iframe").appendChild(i);};
   // Check for browser support of event handling capability
   if (window.addEventListener)
    window.addEventListener("load", createIFrame, false);
    else if (window.attachEvent)
    window.attachEvent("onload", createIFrame);
    else window.onload = createIFrame;
</script>
```

What this does, is have the IFrame load the content after the onload event of the main page. This means nothing will be blocked at all for the main page load. Nice! his means also that when those google bot crawlers go out and read the website, that the main page responds faster and that gets the site a higher search ranking. Are there any other ways to load IFrame and minimize main page blocking? You can also expand on this and do to async load the IFrame and avoid blocking. This is one of those creative ones that really buys some performance regardless of where the IFrame exists on the page:

The wondrous magic of this technique is that with the **<body onload="">** the IFrame has no content initially, so onload of the IFrame fires instantly. Yes instantly! Then, after it has already fired you can create a new script element, set its source to the JS file that loads the content/widget, append the script to the **HEAD**. IFrame content loads without blocking the main page onload! You will see the following behavior:

1. IFrame starts loading before onload of main page
2. Onload of the IFrame fires instantly, because the IFrame content is loaded after onload. So, none of that blocking nonsense
3. Onload of main page fires without interference of the IFrame. The IFrame will not block the main page onload at all because of the **<body onload="">**

So, now you have seen some techniques beyond caching and lazy loading around asynchronous loading of an IFrame without blocking the main page or waiting on dependencies between loading events. Leveraging async or scripts with event dependencies; thus, harnessing the behavior of website events and assure that your pages

Conclusion

Performance automation plays an indispensable role in maintaining the efficiency and effectiveness of web pages. In the modern world, users demand quick, responsive, and reliable websites. This means every second matters. Through automation, the complex and multi-faceted task of monitoring and optimizing web performance metrics like Core Web Vitals becomes streamlined and consistent. It allows developers to identify and address potential issues preemptively, preventing them from escalating into larger problems that could impact user experience and search rankings. Furthermore, automation saves valuable time and resources that can be dedicated towards enhancing other dimensions of a website. In essence, implementing performance automation is not just important, but essential to maintain a competitive edge in the evolving digital landscape.

In the next chapter, we will discuss **Java Virtual Machine** (**JVM**) optimization. It will include deep diving into native code compilation and tuning web services to achieve optimal performance. We will dig into how best to tune the runtime environment.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 7
Java Virtual Machine Optimization

Introduction

This chapter of the book is one of the most important ones. It will introduce you to **Java Virtual Machine** (**JVM**) optimization. In the last chapter, we talked how about to optimize a web platform and the backend for many platforms are services written in the Java computer language. In order to have a fast web platform, the services behind it must also be fast and one of the best approaches to good service performance is JVM optimization. Slow web services running Java means not only a poor user experience but also greater run-time cost.

Understanding JVM optimization and implementing strategies for improvement to reduce run-time costs and improve responsiveness and scale can make a dramatic difference to your online presence. Remember, a faster, more responsive service means that browser interactions and web page load times are faster. Just like web optimization, this not only brings in more user engagement but also improves your SEO rankings. But this is not just about the web, it is also about standalone or **business-to-business** (**B2B**) services and maintaining tight and consistent response times and availability with little standard deviation.

Structure

This chapter covers the following topics:

- Introduction to Java Virtual Machine optimization

- Native code compilation
- Removing overhead/Garbage collector optimization
- Tuning web services

Objectives

In this chapter, we will introduce you to the important concepts of JVM optimization in the cloud. We sincerely hope that this chapter provides you insightful knowledge regarding how to optimize your web services and standalone applications for speed and stability. It involves employing a set of performance strategies and techniques intended to optimize JVMs for the application profile being ran. Let us not wait any longer and dive right into it!

Introduction to Java Virtual Machine optimization

When working with Java, it is crucial to understand that the JVM is really the backbone of your application's performance. The first thing we should always consider whenever we see instability for web services running under load or inconsistency in response times is to first look at the health of the JVM itself. Start by reviewing the major and minor garbage collection times. If **garbage collector** (**GC**) times are increasing and trending upwards, there may be contention preventing cleanup or too much load forcing too much context switching. When JVMs come under too much load, what you will see is garbage collection times trend upward and CPU utilization going upward.

The JVM run-time plays an enormous role in how efficiently your Java applications and micro-services run. This is why optimizing it is of paramount importance. In this chapter, we are going to delve into advanced optimization techniques that will help you get the most out of your JVM. In the world of Java applications and web services, JVM optimization is not merely an option; it is a necessity. We will be discussing how to run Java applications natively and the advantages of precompiling applications. Additionally, we will guide you through the process of code cache optimization and underline the best practices for garbage collection optimization. So, whether you are a novice Java developer or a seasoned Java professional, you are guaranteed to find something valuable in this chapter. The following provides the most important considerations to achieve great performance:

- **Native Java applications**: Compiling applications to native executables or minimal layers of abstraction is a good approach for better performance and enhanced stability. Whether you are using a robust native compilation solution like the popular GraalVM (**https://www.graalvm.org/**) or just precompiling Java code using the **just-in-time** (**JIT**) compiler, going native can really help you achieve some fast speeds. In the cloud world, not only do faster run-times translate to better response times for services, but also much faster startup times for those services.

This allows you to achieve more scale with low amounts of response time outliers during horizontal scaling. Let us consider an example. If you have a Kubernetes pod running a Java micro-service and the pods are configured to horizontally scale from 1 to 20, it means that how fast that next pod scales up matters. Why does it matter? If traffic is surging the application in a short duration, you need the other pod to start fast so that it can start taking traffic before the other active pods become unstable or slow with response times due to being overloaded. If it does not start fast enough, the result will be large swings in response time. If it takes too long and the surge is high enough, it can crash the JVM if the proper threading and queue configurations are not put in place for the service to prevent it. In that case, the service would just become unresponsive without bringing down the JVM itself.

- **Precompiling applications**: Understanding its importance and how to do it effectively. The JIT compiler allows you to fully precompile a Java application. Depending on the size of the application, parameters around the code cache may need to be changed. Code cache is a segment of allocated memory that is used for the native compiled code. The precompiled code stays in this memory space off the heap and can really make the application run fast if it does not have to fall back into Java interpreted mode or have the cache flush and recompile the code on the fly. To do this right, you need to optimize the code cache. If you do not, you may find yourself running too much code in interpretive mode.

- **Code cache optimization techniques**: JVMs have a small default code cache, so to keep the JVM from continually going back and forth between compiled and interpreted modes when your application is running, you need to increase it. This is the first step in JVM optimization. Of course, compiled native code sizes can be hard to predict, so the best technique is to implement the JVM argument **`-XX:+PrintCodeCache`** when optimizing the code cache size. You need to decide if you are going to precompile all code or tune the JVM to compile fewer methods or less lines of code per method. What is the tradeoff? Well, beyond memory usage, the more you tell the run-time environment to precompile, the slower the startup time for the JVM will be. If we start to precompile everything in our application and set a large code cache size, it means that it may take 60 or 120 seconds for our application to start, depending on the size of the application. When it does run, it will be screaming fast generally after startup. For smaller applications, maybe it takes 20 seconds or 10 seconds to startup the JVM. But that is the tradeoff. The more you precompile, the faster the code will run (generally 10x faster than interpreted mode). Profiling or using the JVM argument to print the code cache statistics as the program runs will tell you what the code cache size is when the JIT compiles your application, and it is running. Techniques to avoid memory overflow of a Kubernetes pod or virtual machine and ensure efficient execution of your application is to use a limit to code cache size, a JVM parameter called **`-XX:ReservedCodeCacheSize`**. This option will restrict how much memory the

code cache will use and when the limit is hit, all subsequent not in the code cache will be interpreted after that with slower execution times. Most people do not realize how much code is actually running interpreted due to low default code cache sizes for a JVM.

- **Garbage collection optimization**: Effective strategies to optimize Java's automatic memory management. The garbage collection system comprises an important part of the JVM. Garbage collection sounds like the guy who comes two days a week to clean up your garbage. In this context, it is software algorithms responsible for automatically reclaiming memory allocated to objects the program no longer needs, preventing memory leaks and crashes. Garbage collection optimization is all about fine-tuning the GC's performance. It involves setting proper garbage collection policies for the application running, selecting the right garbage collection algorithm, and adjusting parameters to achieve a balance between memory usage, CPU usage, and application responsiveness.

By the end of this chapter, you will be armed with advanced skills and strategies to optimize your JVM for peak performance. Let us jump right in and start improving your Java applications!

Native code compilation

One of the essential aspects of working with JVM is the concept of precompiling your Java applications. You might be wondering why this is necessary. Precompiling speeds up your application, as the JVM does not have to spend time translating your bytecode into machine code at run-time. As you would expect, though, it is a double-edged sword. When using the JIT compiler to precompile large amounts of code upfront, it means slower startup times. The **`-Xcomp`** option in Java runs the risk of longer application startup times since time will be spent compiling bytecode in advance when the application runs, but it pays off by achieving excellent run-time performance. Using this JVM argument, the HotSpot JIT compiler will compile all source code beforehand with maximum optimization, thus foregoing any profiling that the VM would need to do. If you do not use this switch, the default mode of the JIT is to compile at run-time for hotspots using profiling. The JVM starts up faster, but takes a hit when pre-compilation happens at run-time to compile for the hotspot and put the native code into code cache. The JVM eventually gets faster and faster over a longer period of time before it becomes performant.

Here is how to use the -**`Xcomp`** option:

1. Open a terminal window
2. Type the command: **`java -Xcomp YourJavaApplicationToRun`**
3. Hit Enter to precompile your application

In essence, the **`-Xcomp`** option can significantly speed up your Java application run-time performance. This includes web services where you can achieve great responsiveness with little response time deviation from the JVM cleanup

thoroughly test and benchmark your application with these settings since the benefits may vary based on the nature of your application. The keyword here is optimization.

`-Xcomp` has been an option available in the JVM for a long time. But most people do not know how to use it properly. Remember that there are advantages and disadvantages to compiling all code natively:

Optimization technique	Advantages	Disadvantages
`-Xcomp` option	Faster execution time	Longer startup time

Table 7.1: *Xcomp JVM command line option*

Running Java applications natively implies that the application will run directly on the target platform without the need for an intervening JVM. Many years ago, there were many native Java compilers and de-compilers that you could choose from. But today, there are only a handful to choose from. Some of the most popular were Symantec solutions, where native compilation was built into their IDE, and solutions like **GNU for Java** (**GCJ**). These solutions that were once prevalent are obsolete. For de-compilation JAD was one of the most popular de-compilers that is still even used today. As far as de-compilers the most popular ones today in use are:

- JD-GUI
- Procyon
- FernFlower
- JAD
- CFR

The most popular native compilers are much fewer than in the past. They are:

- JIT
- GraalVM
- Quarkus AOT native compiler

Some of these are hybrid solutions, but when you compile down to machine code, everything just gets better, no other way to describe it. It has several benefits:

- **Improved performance**: Native execution can lead to speedier execution times as it eliminates the overhead related to interpretation and JIT compilation. On average performance will be ten times faster or more than when the JVM runs in an interpreted fashion. code cache optimization is critical when using the JIT compiler.
- **Less memory**: True native solutions use less memory and CPU utilization for many solutions, since it does not require running a JVM and is not trying to compile on

- **Better integration**: Java applications, when compiled to native code, can integrate better with the native operating system and take advantage of platform-specific features.
- **No JVM**: some implementations eliminate the need to install a JVM.

Getting your Java application to run natively is not always straightforward. However, two proven practices are **ahead-of-time** (**AOT**) compilation and GraalVM's native image. We are not going to focus too much on AOT because it was removed in Java 17.

- **AOT compilation and -Xcomp**: In AOT compilation, the JVM bytecode is converted to native code before execution. The advantage is that the execution can be faster, but the disadvantage is that the JVM's dynamic compilation and optimizations are discarded in the process. **`-Xcomp`** will precompile all of the java code and have it run in code cache. AOT is not available in the latest versions of the JVM but **`-Xcomp`** still is.
- **GraalVM's native image**: Using GraalVM, you can compile your Java apps into standalone native executables. These executables can run without a JVM, thus providing faster startup times and lower run-time memory overhead.

Whether to run Java applications natively depends on specific use cases and requirements. While it has several benefits, it may not always be suitable for all scenarios. It is up to you to weigh the pros and cons and decide what approach to take as part of your optimization strategy. Remember that when applications startup slower and then JVMs that are scaling up horizontally will take longer to load. This means spike traffic can become more difficult to manage. Many times, techniques such as lowering the horizontal CPU threshold is one technique that will give another JVM time to startup so it can begin accepting traffic. Sometimes we see folks using memory to scale, which when you have low pod memory in a platform like Kubernetes, and you have JVM max heap settings that are 75% of the heap plus off-heap memory, what you many times accomplish is that all pods scale up all at once. This is not cost effective or a valid run-time optimization.

Precompiling Java applications is always a viable means of achieving optimal JVM performance. This process, also known as AOT compilation or **`-Xcomp`** can be used. Both techniques involve compiling bytecodes into native code before the JVM actually executes them. It provides several benefits:

- **Increased startup speed**: When using tools like GraalVM or AOT, you have code compiled to native and ready to run, resulting in faster application startup. But when using JIT to precompile ahead of time for the java run-time, it means slower startup and faster run-time once started. The JIT compiles first as the JVM starts up before it runs the code. Precompiled code with GraalVM or AOT eliminates the need for the JVM to convert bytecodes into machine code at run-time, resulting in faster application startup.
- **Reduced CPU utilization**: With the JVM not needing to perform JIT compilation during execution, CPU usage is lessened.

- **Optimized performance**: Precompiled applications can perform optimally under resource-constrained environments, as they have a smaller memory footprint and reduced CPU overhead.

However, it is essential to remember that AOT compilation is not a be-all and end-all solution. One reason is that it does not exist anymore after Java 17. When using it in earlier versions, some potential drawbacks include:

- **Lack of platform independence**: When you precompile Java applications using tools like GraalVM, they become platform-specific, negating the write once, run anywhere mantra of Java.
- **Limited optimization**: AOT compilation may not fully optimize certain code paths that would otherwise be improved during run-time JIT compilation. Before opting for AOT compilation, it is important to assess whether the potential performance gains outweigh these drawbacks for your specific application and use case.

The following steps illustrate how you can precompile a Java application:

- Configure your JDK to support AOT compilation. Not all versions of Java offer native support for AOT, so check your JDK's documentation for details.
- Use the **`jaotc`** tool included with your JDK for AOT compilation. This tool turns Java classes into native code.
- Load the precompiled code using a custom **`ClassLoader`** or you can leverage the **`-XX:AOTLibrary`** JVM option.

The specific commands and processes may vary depending on your JDK's version and your operating system. Therefore, refer to the appropriate documentation when attempting to precompile your Java application.

When using pre-compilation techniques like **`-Xcomp`**, the resulting compiled code will reside in off-heap memory in an area called **code cache**. code cache, in simple terms, is a reserved space in memory where Java stores compiled code. Whenever a method is called for the first time, the JVM compiles it into native code using its JIT compiler, and stores it in the code cache for future use. This cache's primary function is to speed up execution time. To optimize the JVM through code cache, you need to be familiar with a few advanced methods. Code cache optimization is an essential part of ensuring that Java applications run at peak performance.

Before delving into the actual optimization methods, it is critical to note that the two primary factors that influence code cache are the **initial code cache** size and the **reserved code cache** size. These two parameters essentially control how much memory is set aside for storing compiled code. Therefore, they directly impact the performance of any Java application. To set the initial and maximum sizes of the code cache, you will always use the JVM options **`-XX:InitialCodeCacheSize=<size>`** and **`-XX:ReservedCodeCacheSize=<size>`**.

The **`-XX:InitialCodeCacheSize`** option sets the initial code cache size during the JVM startup, while the **`-XX:ReservedCodeCacheSize`** sets the maximum code cache capacity and prevents you from overloading a virtual machine or pod in Kubernetes. In both cases, the size is specified in bytes; it can alternatively be expressed in kilobytes, megabytes, or gigabytes with the right designator. For example, if you want to set the Initial code cache size to 150mb, you will write **`-XX:InitialCodeCacheSize=150m`**.

Adjusting these parameters to align with an application need can significantly optimize the JVM's performance. However, it is a bit of a balancing act; if set too low, method calls may become slow due to insufficient compiled code. If set too high, you will waste memory. Hence, it is advisable to do load testing and determine a profile needed for the code cache. You want to figure out the optimal values for your specific application by profiling it under load.

As we have seen, optimizing the JVM is all about applying the right technique to the appropriate situation. It is about weighing up the pros and cons and making informed decisions that improve your application's performance. So, keep experimenting, keep learning, and remember that every little optimization can make a significant impact on your system's efficiency.

Removing overhead/Garbage collector optimization

One often overlooked low hanging fruit optimization is all about the debug code. This critical aspect of responsiveness that needs attention while diving into the world of JVM optimization involves removing debug code in production builds. Debugging in production is rarely done and it wastes a lot of resources and slows down the application. To optimize resource usage and speed, you should consider compiling Java programs using the **`-g:none`** option. You will see similar features in other run-times like AngularJS, NodeJS, etcetera. Here is why this easy-to-implement technique could be a game changer for your Java application performance.

The **`-g:none`** option, when used during compilation in Java, signifies the absence of any debugging information. Debugging information, as might be well-known, is a significant factor utilized during the development and debugging stage of an application life cycle. However, including it in the production stage it consumes unnecessary resources and impacts the running time adversely for an application. In most cases, you will see the run-time code decrease in size by at least 20%. Debugging in production, although an interesting thought, often just means more overhead than a benefit. Remember, every instruction and comment in your code eats up machine time. Yet, these debuggers do not contribute any functional value during application runs in a production environment. So, what makes compiling with **`-g:none`** a preferable approach for JVM optimization in the production environment? Here are some key reasons:

- **Less resource utilization**: As the debugging code is eliminated from the production build, the resource consumption decreases substantially.
- **Reduced compilation time**: The absence of debugging code in the production build leads to a reduction in compilation time as less code is processed.
- **Improved application performance**: With less code to process and fewer resources occupied, the overall application performance experiences a boost.

Implementing the **`-g:none`** option in your Java compilation is simple. Here is a generic blueprint for your command line:

```
javac -g:none MyLowResourceApplication.java
```

In this command, **`-g:none`** tells the Java compiler to skip generating debugging information, leading to much leaner, faster production builds. Remember, JVM optimization techniques require careful consideration and understanding of the application requirements. While compiling with **`-g:none`** can improve performance, it may complicate debugging if problems arise in the production version if you have a high occurrence of debugging in production. So, weigh the pros and cons before implementing this or any other JVM optimization strategy.

Beyond using fewer resources for the application starting out, we are going to delve into a handful of techniques that can optimize JVM through effective garbage collection. By applying these optimization methods, you will see tangible improvements in your Java application's performance:

- **Heap tuning**: A properly tuned heap size can significantly improve the JVM's performance. The JVM heap is where all objects, classes, and JRE classes are placed. If it is too large, it can cause longer GC pause times. Conversely, if the heap size is too small, it can cause frequent GCs, which can also impede performance. The trick is to optimize the heap size to strike a balance in memory utilization.
- **GC selection**: There are several GCs available, such as serial, parallel, CMS, G1GC, ZGC and more. The task is to choose the one that aligns with your application's needs and performance goals. All GCs have their strengths and weaknesses, and their effectiveness can vary depending on the use case. Assessing a GC's performance should comprise several factors like throughput, pause times, footprint, and promptness.
- **Garbage collection monitoring**: It is important to remember to monitor your GC's performance regularly. Tools like VisualVM, Java Mission Control, and others come in handy for this. These tools enable you to understand memory usage, identify memory leaks, and provide insights to tune garbage collection effectively. Additionally, Java provides an option, **`-verbose:gc`**, to log GC activities. This record can serve as a valuable resource when optimizing GC.

In the world of JVM optimization, the term **humongous object** is used to describe any object that is significantly larger than the region size in the heap. As a rule of thumb, if the size of an object is more than half the size of a region, it is considered humongous.

The issue with humongous objects is that they can quickly fill up the heap and drive the GC to work overtime and send CPU utilization through the roof, consequently hurting your application's performance. This is especially a problem if the heap has not been configured correctly, or is not big enough to handle these large objects. Incorrect setup of the heap in the context of managing humongous objects can overload the system and significantly impede performance. Strategies to deal with humongous objects include optimizing the heap size to accommodate them or, even better, refactoring the application source code to prevent the creation of humongous objects by better designing your application's data structure.

The need for managing humongous objects varies depending on your application's design and the type of data you are dealing with. Here are a few techniques:

- **Application profiling looking for humongous objects early**: Use profiling tools to find out if humongous objects are impacting your application's performance. From a coding perspective, minimize the usage of large objects, break up the logic, and make it more granular.
- **Heap optimization**: Consider the load on the JVM and threads being used with the humungous object and set your heap size accordingly. But only if large data structures are unavoidable in your application. If, for some reason, you cannot change the design, then you may have to get creative and restrict traffic coming into the JVM and the JVM threads used that consume the heap. Then, scale horizontally for requests in cloud solutions. The other solution is to scale very high vertically if a lot of traffic is being sent to the JVM for processing.
- **Do not create humongous objects**: Where possible, refactor your code to create smaller objects and data structures.

In the final clause, set large objects to null to suggest that garbage collection runs as soon as possible as an added measure to help prevent overloading of the heap. This is something that you always want to do in the case of humongous objects. It is always a good practice in java code to have a try catch finally block. In the finally block, set references to large unused data structures to null and suggest that the GC run. This will keep your memory utilization lower when the JVM is under load.

Choosing the optimal GC for your Java application will determine its performance. If you are writing a Java Swing application, you are going to want to use the Serial collector most likely and it will perform just fine. But the Serial collector will do terrible for a web service. This collector used to be the default setting for old generation object cleanup for Java version 8. In most JVMs there is a young generation heap called Eden and an older generation heap referred to as **OldGen**, the older generation heap is where long surviving objects are held in the JVM. The young/new generation heap is where new

cleanup both kinds of objects. Beyond Java 8, luckily, the default changed to the G1GC collector soon after that version. This solved a lot of issues for people who did not put in JVM settings for their java run-times which defaulted to the serial collector for a high throughput web service. As you can quickly see, the dilemma lies in selecting from the array of GCs available, each with its unique strengths and weaknesses. But the JVM has evolved significantly since Java 17 with the long-term support of ZGC. In this section, we will discuss the most popular GCs as follows and discus their drawbacks. These include the following:

- Serial
- Parallel
- CMS
- G1GC
- ZGC

We will talk in detail about their individual advantages and drawbacks. The Parallel Collector, also known as a **throughput collector**, aims to maximize the throughput and efficiency of your Java applications. We generally see IT platforms that need high throughput use this one for heap sizes under four gigabytes in size. It utilizes multiple threads to perform garbage collection, hence, it is suited for multi-processor environments. Let us delve into the key benefits of the Parallel Collector:

- **High throughput**: If you need high throughput historically the Parallel Collector has been the one to use. It leverages multiple threads for parallel processing even though those threads are restricted by the CPU cores available. They cleanup java objects that are no longer needed by the application in a parallel fashion. This concurrent usage of multiple threads often results in higher throughput compared to other GCs like Serial GC.
- **Scalable**: As long as the heap size is not too large, the Parallel collector is highly scalable and beloved. For large heap sizes GC times become large. Its performance improves with the number of CPUs, making it well-suited for applications running on servers with multiple cores.
- **Performance**: It follows a stop-the-world mechanism for both minor and major collections, resulting in longer pause times. However, these pauses lead to enhanced performance since all garbage is cleaned in a single sweep, freeing up substantial memory.
- **Efficient**: This collector makes optimal use of CPU resources. It compresses live objects into one end of the heap after garbage collection resulting in improved memory efficiency.

The following table demonstrates a comparison of the various GCs available in Java. It elaborates on their pause time, throughput, and whether they are suitable for manual or automatic management:

Garbage collector type	Pause time	Throughput
Parallel collector	Long	High
Serial GC	Short	Low
CMS	Short to medium	Medium
G1GC	Short	High
ZGC	Ultra-short	High

***Table 7.2**: Comparison of the various GCs*

Choosing the right GC depends on the specific needs of your application. While Parallel Collector might be the right choice for high throughput, you might want to choose others based on factors like pause time or automatic memory management. For example, Parallel Collector has good cleanup times for heap sizes less than four gigabytes in size. G1GC does better for heap sizes greater than 4 gigabytes. This is where this collector becomes beneficial. For lower heap sizes, it is around twenty percent slower than Parallel GC on average. But guess what? ZGC does good at all heap sizes and keeps pause times low. More GC threads can be optimized and balance out the GC cleanup times with resource utilization. The advantages of using ZGC are:

- **Low pause time independent of heap size**: ZGC significantly reduces pause times. Unlike other collectors, pauses with ZGC usually do not exceed 10 milliseconds and are not affected by the size of the Heap or the live-set if the threads are left unrestricted. That being said, achieving ultra-low pause times means consuming more resources. You can balance this. You can restrict the ZGC threads and reduce the resources used and balance out the cleanup time. You will find that ZGC has the lowest cleanup deviation time compared to the other collectors. So, you can tune it for 200ms cleanup, and it will be consistent compared to Parallel, which may vary from 200ms to 800ms for the same application.
- **More response time translating to more throughput**: With its concurrent compacting collector, ZGC helps eliminate long pause times, allowing your applications to run smoothly and efficiently.
- **Scalability**: ZGC is designed to work with terabytes of memory efficiently, hence making your application highly scalable.

By integrating these benefits, ZGC paves the way for enhancing JVM's performance to meet modern applications' challenging demands, refer to the following table:

Garbage collector	Pause times	Performance	Scalability
ZGC	Less than 10ms	High	Up to terabytes

***Table 7.3**: ZGC enhancing JVM's performance*

ZGC's superior performance shows why it is gaining traction among developers seeking optimum JVM performance. Do monitor its impact on your applications, as the best GC is invariably application-specific. A skilled engineer versed in JVM performance optimization will always consider the unique characteristics of the application before selecting a tool or strategy.

The Z GC, or ZGC, is a scalable, low-latency GC introduced in more recent versions of Java. When you want to configure the ZGC for an application, you will need to modify several parameters to ensure optimal performance. Each parameter offers a way to customize how ZGC interacts with your Java program. Below is just a sample of what is available when tuning a ZGC collector in the JVM. This is the basic configuration where you can limit or expand the threads of the collector to lower or increase GC time depending on CPU utilization vs. cleanup time wanted. For example, if we want 5ms for GC cleanup time, it means more CPU, and we need to set **`-XX:ConcGCThreads`** to a high value. If we can live with 100ms garbage collection time, we can set **`-XX:ConcGCThreads`** to a relatively low value to have fewer threads doing cleanup.

Parameter	Description	Settings
`-XX:+UseZGC`	This flag enables the use of ZGC.	It should be enabled.
`-Xmx4G`	This defines the maximum heap size, and in this case, it will be 4 gigabytes.	Set to 4G for a 4GB heap size as an example.
`-Xms4G`	This flag sets the initial and minimum Java heap size. The JVM will not start if it cannot reserve this amount of memory.	Set to 4G to match the max heap size, ensuring the heap does not have to resize during application execution.
`-XX:ConcGCThreads`	This parameter determines the number of threads ZGC will use.	The optimal number will depend on your system's core count. For most systems, the default value will be fine. But this is the setting you use to throttle the GC threads if you want to balance resource utilization with low pause times.
`-XX:+UseDynamic NumberOfGCThreads`	This parameter will dynamically adjust the number of threads based on the load of the JVM to bring down collection times to under ten milliseconds.	Use this setting combined with -XX:ConcGCThreads to restrict threads and balance collection time with resource utilization. Low pause collectors will take up much more CPU and memory if left unrestricted to achieve low pause times. Consider the needs of the application and set thread limits appropriately.

Table 7.4

Using these parameters, your JVM startup command might look something like:

Example:

```
java -XX:+UseZGC -Xmx4G -Xms4G -XX:+UseDynamicNumberOfGCThreads
-XX:ConcGCThreads=100 -jar yourJavaProgram.jar
```

Adjusted settings will require monitoring. Keep an eye on key performance metrics to ensure your application is benefiting from these ZGC configurations. Remember to be ready to fine-tune as necessary.

One essential aspect JVM that you should definitely pay attention to is known as **thread stack size** (**TSS**). This critical parameter can be adjusted by setting the **`-xss`** option for the java run-time that your Java application is running under. By properly optimizing this, you can significantly maximize the number of threads your JVM running the application can support, thus enhancing its overall operational efficiency and reliability.

Essentially, the **`-xss`** parameter determines the stack size for each thread in your Java application. When a new thread is created, a new stack is also created, which comes at the cost of memory. Therefore, a smaller stack size means you can have more threads in your application. However, if a thread's stack size is too small, it can lead to **`StackOverflow`** errors or **`OutOfMemory`** errors, thus disrupting the normal operation of your application. Most java applications can safely run at 256 kilobytes versus the 1 megabyte setting that is the default. This means that you can support four times more stack threads using this setting.

Finding the right balance is incredibly important. By properly tuning your **`-xss`** setting, you can strike a critical balance between memory usage and maximum number of threads, achieving optimal performance for your Java application. Remember that all threads in a process will share the heap. All Java objects in an application, by default, are stored on the heap. Java is a pass-by-reference language, and the objects are not generally, by default, duplicated between methods or threads; only pointers are passed around. The stack holds primitives and pointers to objects mostly when the application runs.

How to set **`-xss`** Parameter?

Let us see how to set the **`-xss`** parameter in Java:

- **Command line**: Use the **`-xss`** option followed by the size of the stack. For example, to set the stack size to 512k, you would use the following command: **`java -xss512k MyTunedJavaProgram.`**
- **Environment variables**: You can also set the stack size using environment variables. For example, in Linux, you can use the export command: **`export _JAVA_OPTIONS=-“Xss512k".`**

What about the **Concurrent Mark Sweep** (**CMS**) collector? While the CMS collector is constructive for specific Java applications, like internal IT department level applications or administrative portals, it does bear some disadvantages. It is crucial to keep these

limitations in mind when deciding if it is the right choice for your IT platform. The CMS collector strives for minimal pause times and can manage large heaps, but it has limitations that might affect the performance of your applications.

The CMS collector is a somewhat reliable collector that performs garbage collection for medium-scale Java applications. It minimizes pause times and ensures efficient resource utilization. However, it is not without its fair share of challenges, including memory fragmentation. This is the largest downside to using CMS, and we would not recommend using it, considering much better solutions are available now. Remember that CMS was around before G1GC or ZGC options were invented. Memory fragmentation is an issue that happens when memory is allocated and deallocated in such a way that it leaves behind unusable spaces in the memory segment. This subsequently limits the amount of contiguous memory space available for larger objects, leading to inefficient memory usage.

The Parallel Collector and ZGC offer alternative solutions to the memory fragmentation issue presented by the CMS collector, and G1GC also does a better job. Think of CMS as the collector to use for medium-sized applications with heap sizes under 4 gigabytes in size. G1GC excels above 4 gigabytes but does not do as well as ZGC when it is tuned properly. Parallel is restricted to performing well below 4 gigabyte heap sizes. So, each collector has its own restraints. Here, we highlight some key drawbacks of the G1GC collector when compared to parallel or ZGC collectors for yielding the best performance.

- **Increased CPU usage**: Instead of emphasizing pause times, the G1GC is designed to use more CPU resources to ensure better overall application performance. This behavior could lead to scenarios where the G1GC collector may cause higher CPU consumption, specifically during compacting phases.
- **Mixed garbage collection latency**: While G1GC is excellent at reducing the length of stop-the-world events, it could also potentially lead to increased mixed garbage collection latency over time with a lot of standard deviation.
- **Parallel or ZGC over G1GC**: These collectors emphasize reducing the latency of each individual garbage collection event, offering potentially much better performance in specific use-cases.
- **Predictive calculation challenges**: The G1GC collector employs predictive calculation to optimize the evacuation phase. Many times, however, this feature can sometimes miss its target, leading to full GC events, which are significantly more time-consuming. Unpredictable full GCs are the largest downside to the G1GC collector.

Why to use ZGC or Parallel collectors over G1GC?

Collector	CPU optimization	Latency	Predictive calculation
G1GC	Increased usage	Potential for increased latency	Inconsistent, may frequently cause Full GC events
Parallel or ZGC	Efficient usage	Reduced latency	No predictive calculation

***Table 7.5**: ZGC vs. G1GC/Parallel collectors*

Keep in mind that performance optimization is not a one-size-fits-all. You will need to analyze your system's needs and constraints to select the best GC for your specific IT platform. The evolution of ZGC has significant flexibility around performance yielding low pause times and great application performance at the expense of more CPU and memory be leveraged. The Parallel collector has great throughput, but higher GC pause times impact the platform's response time.

To always ensure that your application is running at its best capacity:

- **Monitor metrics**: Be prepared to make necessary adjustments based on the performance and profiling data. Make sure that you are regularly profiling your environments to understand the dynamics and correlation between JVM metrics and code running in individual threads.
- **JVM optimization**: Continually optimize the JVM settings and be willing to make the needed changes as the code and infrastructure evolve. This directly influences your system's throughput and workload.
- **Understanding heap size in Java**: The heap is a run-time data area that JVM reserves for dynamic memory allocation. It is the portion of memory where the JVM stores data and variables to operate efficiently. The heap size limits the number of objects the JVM can allocate, casting an influence on your application's performance. There's no one-size-fits-all approach; the optimal heap size depends on several factors, including the number of threads, the kind of your application, and more. However, these general steps can guide you:

    1. **Analyze your application**: Start by understanding what your program needs. Look at what the application does, its memory consumption, and its garbage collection process. Profile the application and review resource usage as a starting point. For example, how much memory is each thread using for a request. If you do not have profilers enabled to do this, you are subjecting yourself to potential poor performance down the road. A well-optimized JVM should not be able to be overloaded if the proper thread restrictions are in place.

    2. **Default size**: If you have no specific size requirements for a desktop Java application, utilizing the JVM's default value could be advantageous. The JVM's default size is buffered, considering general use cases. Thus, it is not a bad starting point for a Swing or JavaFX

Profile the application and adjust beyond the default settings if needed. Never use default settings for web services, ever! For micro-services, we have found that for most applications 512m maximum and 512m minimum heap size to be a good average minimum starting size for simple services that need to scale to fifty requests per second, with service response times under 1 second, where memory usage for a thread is less than one megabyte for a single request.

Remember, that to achieve the best efficiency in the JVM, you set the max heap and min heap settings to the same value. But if you have poor coding practices with memory leaks or a lot of humongous objects, then we would recommend that the min heap size be seventy five percent of the maximum heap size. You will find that the JVM is more forgiving. If you do not do this, you will find that the JVM will protect itself and force full GCs that you cannot predict for large durations.

3. **Optimize**: Monitor your application's performance and resource utilization closely using profilers and observability tools continually. If you notice any inefficiencies or issues, adjust the size accordingly. Remember, this is an iterative process; continuous optimization is key to maintaining the best performance.

What are the most common entities for a Java application running on the heap. If you do a heap dump and do some analysis using tools like VisualVM or MAT, you will see the following:

- **Objects**: As stated earlier, all created objects of your Java application reside in the heap space. When objects are no longer needed, the GC removes them to free up memory. Generally, objects are the most significant occupants of the heap space.
- **Class instances**: Instances of classes also consume heap space in Java. Class instances are essentially objects themselves, but it is also worth noting that each instance of any class will add to the heap's memory usage.
- **Strings**: In Java, if you want great performance, then optimize how strings are used. We will cover Java language optimizations in the next chapter. Strings are also objects and, therefore, will occupy heap space. Moreover, String objects are typically numerous and can significantly impact memory utilization due to their immutable nature, causing new objects to be created with every modification.
- **Arrays**: Arrays in Java are also objects. Whether it is an array of primitives or an array of object references, arrays dwell within heap space and add to its overall memory usage. Memory usage can increase quickly with large multi-dimensional arrays.

Off-Heap storage is the portion of JVM memory that is used for storage purposes outside the regular Java heap area. It directly interacts with the computer's memory at a lower

Entities typically stored Off-Heap:

- Thread Stacks (pointers to code/objects running in heap)
- JIT code cache (native compiled objects)
- Method area
- Direct ByteBuffers
- MetaSpace (or PermGen in older versions)

Before we delve into the details of optimizing your JVM for better performance, it is important to define and understand more about the JIT compiler and the code cache. The code cache is a system of storage for the JVM that exists Off-Heap for storing native compiled code. But what exactly does that mean, and why is it important? Let us explore that in depth.

The JVM, behind the scenes, does a significant amount of work to make your application performant. One of the JVMs JIT HotSpot compiler's responsibilities is to compile commonly used methods into native code and put the results in the code cache. Remember that it does this often so if you have a code cache that is continually flushing and recompiling, this can be a detriment to performance. Sometimes referred to as hot methods are methods that are slower when an application runs in the JVM. When you see something running slow at run-time, do not just consider I/O and the code itself, also consider what is happening around the JVM contribution to latency itself through recompilation and the code cache sizing/operations around cache flushing.

Off-Heap here refers to the storage of information outside of the main memory allocation area for your running Java programs, otherwise known as the heap. This is where JVM stores its compiled code. The utilization of Off-Heap storage can be instrumental in maximizing your Java application performance as it helps efficiently manage the compiled code. Below are key factors that make Off-Heap storage of code cache important:

- **Separation of concerns**: By keeping the compiled code off the heap, the JVM ensures that garbage collection processes will never interfere with the compiled code.
- **Maximizing heap space**: Storing the code cache off-heap will save a great deal of heap memory, meaning more room is left on the heap for your program's data. This can help optimize the performance of your application.
- **Control over code cache**: When code cache is stored off-heap, developers have more control over it. They can tweak its size, monitor it, flush it, and take more direct actions to optimize JVM performance.

Having understood the importance of Off-heap code cache, it is time that we dive into more advanced techniques on JVM optimization. The first thing to know is that heap size also influences JIT compilation. JIT plays a crucial part in achieving high performance in Java applications by compiling bytecode into native machine code at run-time, leading

in suboptimal performance. These do have a correlation. The optimal heap size setting is inherently dependent on the specifics of your particular use case. Regular monitoring and adjusting, factoring in the role of elements like JIT, is imperative to ensuring your Java application's peak performance.

Since the JIT compiler uses a portion of memory known as the code cache to store the native machine code it generates, ensuring this area has sufficient size can be critical for optimal performance. Tuning the code cache can have a significant impact on the performance of your Java applications. If the code cache becomes full, it may result in newly compiled methods being evicted, leading to frequent recompilation and degraded performance or too much code being interpreted or a combination of both depending on the optimization settings and dynamics. There are techniques you can utilize to optimize the performance of the JIT compiler and, by extension, your Java applications:

- **Understanding the compilation threshold**: Adjusting the compilation threshold is a technique that is often overlooked, but it is incredibly significant in optimizing your JVM. The JIT compiler does not compile all code immediately. It will wait until a method is called a certain number of times, which is referred to as the compilation threshold, which is set before compiling it to native code. Tuning this threshold value is an important optimization technique to balance optimization with code cache memory resource utilization, startup JVM times, and more.

  This concept is pivotal in JVM tuning and can greatly impact the performance of an application running on the JVM. The JVM has been designed to strike a balance between the time spent in compilation and the speed of execution of the compiled code. A higher compilation threshold means that methods are compiled less often, whereas a lower one means they are compiled more often. Compiling less often can often result in much improved run-time performance with decreased CPU usage. However, setting the threshold too high could lead to an increase in the invocation overhead, which could potentially slow down your application.

- **Profile-Guided optimization**: To understand the dynamics around how compilation thresholds impact performance and resource utilization, run your application with a profiler enabled, which provides detailed information on which methods are being called most frequently. Remember that these are the methods that the JIT will target by default. Beyond using expensive profilers or observability tools, the JVM itself has many parameters to give insight into compilation, GC collection, and what is going on internally with the JVM.

- **Startup time optimization**: This is a balancing act when it comes to optimization. It can be done by adjusting JVM parameters to either prioritize quick startup time or slower startup time with maximum performance once running.

Let us dig deeper into optimizing the JIT and establishing the right Compilation Threshold. The JIT tuning parameter, compilation threshold**,** can be adjusted using the **`-XX:CompileThreshold`** JVM argument. The default threshold is usually set at 10,000,

setting the compilation threshold, consider the nature of your application. If it is a long-running application, a high threshold might be best. For applications that require quick startup times, a lower threshold may be more effective.

Remember, every Java application is unique, and there is no one-size-fits-all setting. Therefore, thorough testing and a proper understanding of the application's behavior are crucial when tuning the compilation threshold. The best approach is to monitor, measure, adjust, and repeat until you find the optimal setting.

Another optimization step we can take when we are constrained on resources is to flush the code cache. Flushing the code cache is one critical operation that aids in the optimization of JVM, especially in situations where the code cache is close to being full or is already filled. Clearing or flushing the code cache can effectively remove the unnecessary clutter, making room for newly compiled methods. This process of removal supports optimal performance by reducing the chances of frequently evicted and recompiled methods. You can enable the automated flushing of the code cache by specifying the following JVM option during application startup: **`-XX:+UseCodeCacheFlushing`**.

Set the code cache flushing threshold using **`-XX:ReservedCodeCacheSize`**. Java will begin flushing the cache once usage exceeds this set limit. Monitor your application performance steadily. If you notice frequent full code cache events which will hurt your performance, consider increasing the reserved code cache size.

You can also manually flush the code cache using tools like JMC. Here is how:

- Open your JMC.
- Connect it to the JVM of the running application.
- Under the **MBean Server** tab, go to com.sun.management > HotSpotDiagnostic.
- Run the flush operation.

The default settings for the size of the area where natively compiled code is stored are relatively small unless the size is defined. The larger the area, the more native code that can reside there. When there is not much room, the cache can be flushed, and recompilation can happen for different codes used at different times using the parameters below:

Parameters that must exist as part of the JVM startup	
`-XX:+UseCodeCacheFlushing`	Enables the flushing of the code cache
`-XX:ReservedCodeCacheSize`	Sets the flushing threshold for the code cache

***Table 7.6**: Code cache optimization settings*

Remember, flushing the code cache effectively optimizes JVM performance, but must be carefully managed to avoid the overhead of frequently evicting and recompiling methods. Always keep tabs on your performance metrics when adjusting these settings. It is also important to note that JIT compilation and garbage collection are two JVM actions that

is crucial for optimal performance. Keep a close eye on the performance metrics of your application after making any adjustments.

Tuning web services

Beyond JVM optimizations, such as compilation thresholds and code cache optimizations, one of the key factors for a well-performing platform is to tune the web services in the platform to the JVM settings. You want to optimize the inbound traffic and align that traffic to the JVM settings to ensure optimal run-time performance.

To do this, you need to tune the Tomcat or Netty connectors in your spring micro-services setup. Tomcat is a fundamental element in the execution of your Java application in the JVM. By effectively tuning the Tomcat connectors, you can manage to not overload JVM threads, and improve the performance of your application significantly. Let us dive deeper into tomcat optimization techniques. First, you want to look at the following settings:

- **Setting the maximum thread count**: You can limit the creation of excessive threads, which can overload the JVM, by setting the **`maxThreads`** attribute in the connector configuration of Tomcat. A good starting point might be 200 threads.
- **Connection timeout**: The **`connectionTimeout`** attribute defines the timeout in milliseconds to wait for a connection to be established. By adjusting this value appropriately, you can prevent long-lived connections from overwhelming the JVM.

  Example1

  ```
  <Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000"
  redirectPort="8443" maxThreads="200" />
  ```

Next, you want to look at tuning the thread pool. A thread pool manages a pool of worker threads. Tuning the size of the thread pool can significantly enhance the performance of your JVM. The configuration involved for the Tomcat connector is as follows and involves defining attributes that can be used to keep a certain amount of threads always available, set how many threads can be processed before waiting occurs for a request, and how long threads will remain open before being shut down if a request does not come. This is just a small sample of what is available to optimize.

Property	Description
`maxThreads`	The maximum number of request processing threads to be created by this Connector, which therefore determines the maximum number of simultaneous requests that can be handled.
`minSpareThreads`	The minimum number of threads always kept running.
`maxIdleTime`	The maximum number of milliseconds that the pool will keep spare threads open.

Table 7.7:

Always remember that these settings may vary depending on your application's specific requirements and use case. Adjust these settings carefully and monitor the results to ensure the best performance. When running spring micro-services in a Tomcat environment, a few key configurations can help optimize performance. You can find these settings in the **`server.xml`** configuration file.

To achieve great performance, it is all about balancing traffic handling capabilities for the JVM settings. Let us delve into a critical setting called **`acceptCount`**. The **`acceptCount`** setting is often deemed as a magic configuration option that balances incoming connections to your service. But what exactly does it do? **`acceptCount`** is the maximum queue length for incoming connection requests when all possible request processing threads are in use. If a request cannot be processed immediately by a worker thread, it goes into this queue. If the queue is full, the connection is refused. This essentially acts as a buffer between your service and the outside world. Here is an example of the syntax in a configuration file called server.xml to setup **`acceptCount`**:

Example:

```
<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000"
acceptCount="100" />
```

Remember to balance **`acceptCount`** to your service's needs. Too low, and you might not be able to handle sudden influxes of connections. Too high, and you might consume more resources than necessary. Start with a moderate setting and adjust based on your observed performance. Running a spring micro-service efficiently involves striking the right balance in configuring your connections. The **`acceptCount`** plays a major role to optimize your service's behavior under varying load conditions.

To optimize your service's performance, it is crucial to tune the **`acceptCount`** and **`maxThreads`** at a minimum. Begin with a moderate a**`cceptCount`** and **`maxThreads`** setting. For example, start with 100 for max threads and 20 for **`acceptCount`** as a guide. Observe your service's performance under varying load conditions. Pay special attention to how well it handles sudden peaks in connection requests. Based on your observations, adjust the **`maxThreads acceptCount`** either upwards or downwards. While ensuring your service's capability to handle connection requests, safeguarding against high resource consumption is equally important.

Too high of **`acceptCount`** and **`maxThreads`** settings might consume more resources than necessary. Always remember that when optimizing software well and finding starting points, it is all about mathematics. If you know that a single connection request coming into the Tomcat connector uses 5 JVM threads and 10 megabytes of memory and the response time back to the connection is 1 second, then you can easily determine how many connections your JVM can handle and what heap settings need to be using simple math.

Create settings using mathematics when you can, and then use load tests to verify and fine-tune. You need to balance the requests coming into the service or increase the resources the JVM uses. But keep in mind, if the service is part of Kubernetes or another horizontal

scaling container manager, then you need to be able to reach the threshold around CPU to scale to the next Pod. You want to use the queue **`acceptCount`** for spike traffic versus spinning up more pods horizontally, but you also want to be able to reach the threshold for scaling for normal operation by configuring **`maxThreads`** properly. Remember to always make sure that these settings cannot ever crash the JVM. Do not allow more threads to run or connections than the JVM will support with its memory settings. Make sure that you allow just enough threads into the JVM from the connector to reach the scaling threshold.

Remember that **`acceptCount`** is an attribute that determines the queue size for requests coming into a service. It needs to be optimized. It can greatly increase the scale of the service at the expense of some additional latency. For spikey traffic, it can be a godsend because additional horizontal scaling may no longer be necessary, which increases run-time cost and many times latency because a new run-time environment needs to be spun up. Below are the advantages and disadvantages of a**`cceptCount`** sizing.

`acceptCount`	Effects
`Low value`	May lead to service being unable to handle sudden influx of connections
`High value`	May lead to higher than necessary resource consumption

***Table 7.8**: acceptCount queue optimiztion*

Therefore, constantly observe the performance through profiling and adjust the **`acceptCount`** and **`maxThreads`** accordingly. Your ultimate goal should be to achieve a balance that caters to your service's needs efficiently without consuming more resources than necessary. There is so much out on the web about tuning Tomcat or Netty and the JVM. The important thing to remember is that the default for a spring service using the Tomcat connection is 200 max connections and 100 for an **`acceptCount`** queue depth. If you do not optimize these settings or use the default, you are not efficiently processing requests to your service and are wasting money. Here is a simple example of a configuration for Tomcat.

- **`server.connection-timeout=10s`** / /sets timeout for a connection
- **`server.tomcat.accept-count=100`** / /queue
- **`server.tomcat.max-connections=1000`** / /connection request max
- **`server.tomcat.max-threads=200`** / /max worker threads
- **`server.tomcat.min-spare-threads=100`** / /min threads to keep running

There are many configurations for the spring Tomcat connector, and the Netty connector varies slightly, but these are the most important ones to pay attention to when tuning requests coming into service and interacting with threads in the JVM. Add a lead-in statement here for the following:

- **`server.connection-timeout`**: Time that connectors wait for another HTTP request before closing the connection. When not set, the connector's container-specific default is used. Use can a value of -1 to indicate infinite timeout.
- **`server.tomcat.accept-count`**: Maximum queue length for incoming connection requests when all possible request processing threads are in use.
- **`server.tomcat.max-connections`**: Maximum number of connections that the server accepts and processes at any given time.
- **`server.tomcat.max-threads`**: Maximum amount of worker threads in server under top load which translates to the maximum number of simultaneous requests that can be handled.
- **`server.tomcat.min-spare-threads`**: The minimum number of threads always kept running. This includes both active and idle threads.

Conclusion

In conclusion, JVM and web service optimization are critical for tuning IT platforms for great performance and scale while keeping run-time costs low. In this chapter, you learned about the different GC methods, compile thresholds, code cache optimization, and the importance of slimming down the production deliverables to run faster. These steps all contribute to having great running IT platforms. The chapter stayed away from the confusing detail that the JVM documentation and Spring documentation provide and focused entirely on what was most important for optimization. The goal of this chapter was to shed light on the interactions between the JVM and web services and what gives the most bang for the buck around optimization.

The goal was to give you some thoughts around the most important considerations and enough knowledge to want to move further into the topic. There are at least one hundred different settings for the JVM and spring services to fine-tune them beyond what is provided here. We encourage you to dive deeper into JVM and spring vendor documentation and try to fine tune your services further and build upon this knowledge. The vendors documentation can be referenced for all of the settings with a description of each. This chapter focused on the alignment between JVM resources and incoming requests and what are the most important aspects to consider for tuning the JVM to be successful. In the next chapter, we will talk about how to optimize the Java language!

Chapter 8
Java Code Optimization

Introduction

Java source code optimization will improve the efficiency and performance of a Java application or service. It involves augmenting code to run faster with fewer resources, providing the same functionality as before the code optimization took place. You must be proficient in Java to understand this chapter. It is important to have a good grasp of how to work with different variable scopes, data types, operators, loops, functions, and control statements. If you do not have this knowledge, this chapter will not make much sense to you. This chapter will go deep into software coding and discuss some software design principles, design patterns, and considerations around making platforms fast.

Understanding Java code optimization and implementing strategies for improvement to reduce run-time costs and improve responsiveness and scale can make a dramatic difference to your customers. In this chapter, we will continue to talk more about how the **Java Virtual Machine** (**JVM**) and code interplay with one another for optimization and talk about all of these considerations. This will tie together the important concepts around the creation of fast platforms.

Structure

This chapter covers the following the following topics:

- JVM and code optimization relationships
- Multi-threading and asynchronous programming
- Creating fast code through best practices
- Reusable design patterns for high performance

Objectives

In this chapter, we will introduce you to the important concepts of Java source code optimization in the cloud and talk more about the run-time environment as well. We will go even further with more depth and extend the last chapter by discussing how code and the run-time environment work together to achieve optimized code. Design patterns will be discussed. and we will provide you with good third-party sources of information around designing high performant platforms. This, we hope, will help you expand your knowledge of the key concepts discussed.

We sincerely hope that this chapter provides you with insightful knowledge regarding how to optimize your Java code and standalone applications for speed and stability. To achieve scale and great usability we must optimize and employ a good set of test and optimization strategies to build great IT platforms. Let us get started!

JVM and code optimization relationships

In order to have a fast IT platform, the Java web services incorporated into the solution must be fast. One of the best approaches to good service performance is JVM and Java code optimization. A good part of proper coding is using small, optimized, fast classes and lower-level objects with less abstraction layers versus large frameworks. Taking this approach and leveraging the proper design patterns will ensure fast code with less CPU and memory consumption.

As part of the high-performance distributed design, it is important to ensure minimal traffic goes across the network. If you have a four-lane highway and you send too many cars, those cars will slow down once all lanes are full. The same thing happens with a fixed level of bandwidth and connections across network switches and software components in a distributed architecture. The goal should always be to minimize the number of connections and requests going to each layer and leverage caching or pooling in each layer along with efficient parallel processing of threads. You also want to leverage and reuse existing connections already connected from the source to the destination service across each layer as much as possible.

Consider keepalives, connection pools, minimal balancing rewrite rules, minimal redirects, buffers that align with network conversation sizing, compression, minification, etcetera. If you are leveraging a reverse proxy like Apache or Nginx, look closely at memory and disk caching capabilities there and the ability to override headers for forced browser caching. Reverse proxies can add a lot of power to your distributed platform when it comes to performance if leveraged properly. Do not fall into the trap of removing them thinking that is one less call across the network. We have seen this too many times over the years and it is always a mistake. Reverse proxies are powerful tools that provide a centralized way to control and manipulate traffic, caching, centralized response header control, HTTP 2 multiplexing, HTTP 3 UDP type of protocols, and timeouts. They also have an unbelievable number of plugins available to minimize, compress, and manipulate traffic and improve performance. The power to improve performance with reverse proxies is something that you will always want to have available to you for your IT platform. They are much more powerful when it comes to helping with performance than the abilities currently built into most balancers and API gateways. These components of a distributed architecture also have their place, especially when hardware driven.

In the preceding network highway analogy, this means that you want to use trucks and send them less frequently to keep the highway from backing up. In the context of a high-performance distributed software design, it means making coarse-grained requests and taking a lot of consideration in designing high-performance web service contracts, along with the data structures being passed across the network. As a developer, it can be tempting to use remote objects like local objects for an application and use them the same way with fine granular calls. For high performance, you must always treat remote objects differently and consider how much information is going across the network, the connection times if those connections are active, whether the timeouts are set properly, etcetera. When payloads for services or web traffic are large, compression also becomes your friend.

The optimization goal for any platform is always to minimize the number of **Domain Name Service** (**DNS**) lookups, **Secured Socket Layer** (**SSL**) handshakes, balancer requests, reverse proxy requests, and pod container level requests coming into Kubernetes and across the distributed platform. It is also important to consider software resiliency in every design or coding decision. What is going to happen if the downstream service is slow? What is going to happen if a request hangs up? What is going to happen if the downstream service is down? So, redundancy and high availability, retries, efficient error handling, timeouts, and reprocessing queues are essential along with user friend error messages to prevent users from continuing to press a button over and over again waiting for a response, overloading your platform in times of duress.

Something to note when it comes to retries: if a service fails in one region, retry against the other available region. Never have more than 3 retries when the platform degrades; if everyone on the platform is having an issue, that means that now, 3x times the connection volume is occurring on the platform. Thus, you need to load test for 3x the volume and take this into consideration. If you had retried 10 or 20 times, you would likely have

Caching also plays a big role in reducing network traffic, being able to pull from a local application cache versus taking a trip across the network. Consider what can be precached as part of a container starting up beyond enterprise-level caching that requires requests across the network. The fewer requests going across the network, the more resilient the platform and also the more consistent the response time for the application with less standard deviation.

Beyond reducing CPU and memory resources, I/O is one of the largest contributors to latency. This refers to I/O in the service, such as interaction with other web services, databases, downstream platforms, writing to disk, etcetera. Thus, using design patterns that optimize I/O interactions along with minimizing request blocking when possible is the key to high performance and being able to absorb downstream dependency backups. Slow web services running Java means not only a poor user experience but also greater run-time cost.

Optimizing your Java code and JVM may seem like a difficult endeavor, but we are here to assure you that when given the right knowledge, it does not have to be. Successfully optimizing your code for better performance opens up pathways to exceptional results, including better response times translating to a better user experience. Compiling and running that code with JVM optimizations discussed in the last chapter and code optimizations in this chapter will ensure great performance.

This chapter will guide you through practical strategies and patterns for optimal Java coding, providing you with the tools you need to craft the most efficient source code that you have ever written. Remember that Java performance optimization is not just about speed, it is also about building resilient platforms with efficient code that leads to lower support costs. When creating efficient, repeatable patterns for performance and code reduces the amount of code that needs to be maintained in almost all instances that we have seen. Well maintained code maintains performance much longer across release cycles than the spaghetti code alternatives where poor practices are followed.

Compile threshold

First, let us tie together JVM and code optimization. In the last chapter, we talked about the compile threshold, which determines when the code is compiled to native code and is put into code cache. This threshold controls the number of times a method needs to be executed before that compilation happens. To go a little deeper here, there are different types of optimizations around code compilation and different levels. The three primary ones are the client-side, referred to as *C1* (compiler one), the server-side, known as *C2* (compiler two), and the tiered compilation of code. The client-optimized code will start very fast but not perform very well once started. It does a very high-level compilation in favor of faster startup time versus longer profiling and more native code generation. The server-side optimization compilation is the fastest. It will be very fast once running with more native code compiled and put into code cache, but take a longer period to startup. It

Tiered compilation is popular because it is the balance between startup time and speed once up and running. The downside is that it takes up much more memory and resources to compile the same methods as the other approaches; achieving optimization better than *C1* but not as good as *C2*. We would encourage you to leverage what works best for your particular platform.

The JVM parameters for different compilation approaches are generally -client, -server, and **`-XX:+TieredCompilation`**, which is the one that allows the JVM to leverage multiple compilers for different methods based on the frequency of the code being called and uses a lot of memory. Some versions of JVMs have different parameters that you leverage for setting the compilation threshold, so be aware of that. You always need to determine if startup time or fast code is what you are looking for. It is difficult to have both due to the nature of precompiling or compiling on the fly.

Once in code cache, there are three segments of memory used, one is for JVM internal code and the others are for *C1* and *C2* compiled code. If you want to profile and determine what you need, you can optimize the use of resources and configure these segments using **`XX:NonNMethodCodeHeapSize`**, **`-XX:ProfiledCodeHeapSize`**, and **`-XX:NonProfiledCodeHeapSize`**. Something to always remember is that when using **`TieredCompilation`**, 4x the resources will be used in the code cache compared to the other methods. Sometimes, it may be better to set thresholds lower in Kubernetes for CPU utilization and startup new pods sooner, take the longer startup time when it comes to micro-services, and leverage server-side compilation that is faster and uses less code cache. Something to consider depending on the IT platform.

Native compilation into code cache

The three of the most important optimizations beyond when to compile or how much to compile into the code cache discussed in the last chapter are -XX:+AggressiveOpts which creates internal optimizations for the JVM, and -XX:+DoEscapeAnalysis, which the JIT compiler leverages to determine the scope of variables. If a variable is found to be used only within a certain method, the JIT compiler can optimize the code by allocating the variable on the stack instead of the heap, which can significantly improve performance. Another important JVM run-time optimization is to leverage the -XX:+UseCompressedOops argument, which enables the use of compressed ordinary object pointers. This will reduce the memory footprint of your application without significantly impacting performance. This is especially useful for applications that use a large amount of memory.

If you leverage these optimizations in addition to **inlining**, you can achieve some great performance. Inlining is when the JIT takes code previously in methods and inlines it all together to eliminate the call overhead. JVM parameters like XX:FreqInlineSize will determine the number of bytecode instructions to inline for a method. We encourage you to review all of the detailed compiler and JIT run-time options, which can be tuned for best performance. You can go extreme for fast code and create a large amount of inlining

or a small amount depending on balancing compile time, startup time, and run-time performance.

You can achieve extremely fast run-time performance, but you need to always consider startup time or time that the JIT spends on the fly trying to optimize the code. Balancing startup time and minimizing the amount of time the JIT has to recompile due to code cache being too small are the best optimizations that you can make. If you enable the Continuous Profiler in Datadog, an application performance monitoring tool, as an example, you can see when the JIT is recompiling and the latency is incurred. If you are in constant recompile mode, your JVM is not optimized and you are hurting the performance by recompiling sometimes as much as running interpreted code. Many other profilers do this. The **Java Flight Recorder** (**JFR**) can provide you with a great deal of information. You can also leverage what the JVM provides you by default using JVM run-time parameters to gather information, such as **`-XX:+LogCompilation`**, to see the compilation events.

You can also leverage verbose parameters such as -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:PrintAssemblyOptions=hsdis-print-bytes to get a deep view into the compiled JIT methods; for those folks that like to look at assembly code. Beyond optimization after the fact for code, there are also things we can do to improve the code itself to make it run efficiently and also help the JVM optimize itself further. We will broaden our scope to cover areas every Java developer should consider. We will talk about identifying fast or slow Java classes, understanding how to optimize Java Exceptions, and learning techniques to speed up your code using asynchronous code and multithreading. We will talk about how to effectively use memory and reduce the impact on the CPU. Some broad level items that improve performance are as follows:

- **Slow versus fast Java classes**: The first step when it comes to code optimization is understanding which classes offer optimal speed and which are slower. Determining which need to be thread-safe and which do not need to be and minimizing synchronization as much as possible.
- **Java exceptions**: One of the most important concepts to understand is that exception handling is critical in Java programming. It has one of the largest impacts to performance, especially when it comes to instantiating exception classes and taking stack trace snapshots repeatedly. You want to always reuse exception instances when you do not require a stack trace and have your own error numbers and messaging, when possible, versus taking a snapshot of a hundred-line stack dump.
- **Asynchronous code and multithreading techniques**: When you run code asynchronously, it does not block a synchronous thread, allowing it to respond almost immediately to the caller. Multi-threading allows for multiple units of work to all happen in parallel. One of the most important things to learn from a performance perspective is how to leverage multiple threads and multiple asynchronous threads to accomplish work. When you can leverage thread pools of previous spun up threads created during

the thread spin up overhead and even take performance higher. Remember that the more threads that are spun up, the more context switching the CPU has to do between those threads, so you do not want to multi-thread slow code, you want to always optimize the code to minimize the number of threads the CPU is switching to at a given point in time.

- **Efficient memory utilization**: How applications are coded ties into memory allocation needed and how it is cleaned up from the heap. Poor management of memory resources can lead to slow code. We will delve deeper into how you can use memory more efficiently.
- **Reducing CPU times**: Keeping your CPU load lower can significantly boost your application's performance.
- **Minimizing object instantiation**: One of the largest impacts on CPU time is instantiating objects. If you are frequently instantiating objects in loops or in frequently called methods under load, this can slow down an application. You want to pass in already instantiated objects into routines and reuse instances when you can. Instead of instantiation, using an object and throwing it away, re-leverage it!
- **Pre-size collections always**: When using collections incrementally, increasing allocation of memory over time means slower performance and overhead. You want to instead pre-size objects as much as possible. Remember that dynamically sized collections seem very neat but that continual memory allocation is expensive.
- **Program at the lowest level**: Many times, we leverage libraries or classes out of convenience when we do not need to. For example, using the URL class and associated classes versus leveraging bulky libraries like **`HttpClient`** or third-party frameworks like Rest Assured, Jsoup, etcetera. will make your platform scream with performance. We cannot emphasize this enough as a practice. Do not use frameworks or classes where you leverage a small subset of the benefit it provides. Leverage primitives where you can in Java versus objects and leverage objects and object arrays versus large collection classes when you do not need the overhead. The lower level you code to, the faster your code will load and run! Learn to be an advanced Java programmer.
- **Use reference classes**: These tie into JVM optimization because they let the garbage collector know that when the heap gets full, it is fine to go ahead and eliminate them because they will be recreated if needed. The Java reference class has an important capability when it comes to memory optimization. It serves as a base class for subclasses like **`WeakReference`**, **`SoftReference`**, and **`PhantomReference`**, each class having unique behaviors concerning garbage collection. These classes not only help with performance but also ensure that no memory leaks occur for memory-intensive applications. For example, if you are creating a cache, you can leverage a **`WeakReference`** array and if memory starts getting tight for the JVM, it

will cleanup data as needed. Then, you can use the design pattern of going back to the source to create it if needed. For example, let us say you have a simple cache where the key is the SQL statement for a database, and the **`ResultSet`** is the value. When running code, you check the cache first, and if it does not exist, you go back to the database and recreate the cache element in memory. Using these classes, when possible, assures great memory management.

- **Leverage parallel streams**: A parallel stream is split into multiple substreams that are processed in parallel by multiple instances of the stream pipeline being executed by multiple threads. The substream thread results are combined to create the final result.

Java has many unique features that make it a popular choice for developers. It became popular because of the code once and ran everywhere. You could create applications that ran on every operating system and almost any device. But it has its strengths and weaknesses. You need to know what these are if you want to write code that not only works but works fast.

Efficient data structures

An important coding pattern in Java for great performance is to remember to always leverage efficient data structures. Choosing the right data structure for a specific task can greatly impact the performance of your code. For example, using an ArrayList instead of a LinkedList for operations that require frequent random access can significantly improve performance. Similarly, using a HashSet instead of an ArrayList for data checks can provide faster lookup times. It is important to understand the characteristics and trade-offs of different data structures to make informed decisions and optimize your code. What we see today is many developers looking for even faster collections than what Java provides. These include third-party solutions like *Eclipse Collections*, *FastUtil*, and *Google Guava libraries*. These solutions have powerful features and great memory efficiency.

However, before we delve too deep into code and our class options, let us cover some basics. Let us talk about immutable objects. If you have coded in Java for a while, you have undoubtedly come across these. If you have not, this is the path to code efficiency. Immutable objects are Java objects whose state cannot be modified after they are created. This special aspect of immutability has some great benefits when it comes to the reliability and efficiency of your code. For example:

- **Data integrity**: The data's state remains unmodified throughout the execution of the software. This consistency assures the robustness of your application. No other part of your code can modify the data of an immutable object.
- **Thread safety**: Immutable objects are thread-safe. Their state is not affected by any changes in other threads. You can share and reuse immutable objects across different threads and never worry about synchronization. This results in faster applications.

Java has great asynchronous and multi-threading capabilities. But when multi-threading applications, you generally have to leverage synchronized methods to keep threads from overwriting each other's information. This thread safety, though necessary, not only slows down applications, it means that the developer has to be very careful in their software coding to not create bugs. Immutable objects, being thread-safe by default, makes your life so much easier without needing to worry about the complexity of synchronization or data issues which can be very difficult to track down.

When classes are not thread safe

Data races can occur. This event happens when multiple threads try to change data simultaneously, causing inconsistent results in code and data. Imagine if you were updating a variable that controlled a loop as a simple example, and another thread was updating that same value, putting your software run-time into infinite chaos.

Code synchronization issues arise when we need to ensure that only a single thread accesses a particular block of code at a time. Developers handle this with explicit synchronization using keywords or locking objects, adding complexity to the code. Immutable objects eliminate these issues since they can be freely shared amongst threads without risking data inconsistency. Using immutable objects can lead to more robust, thread-safe, and performant Java applications, saving you a lot of support costs.

With immutable objects, there is no need to worry about things changing; you can pass around your immutable object to other parts of your code, and none of its parts will be altered. This makes your code solid, eliminating the chance of unexpected side-effects. Immutable objects can provide some significant performance enhancements. This especially comes into play with:

- **Memory**: Immutable objects are lightweight on memory.
- **Cache**: They can be cached and reused across multiple threads without the need for memory and CPU intensive copying.

Now that you know the benefits of using immutable objects in parallel computing, you can write faster and more efficient Java applications. Their robustness can help save your time and prevent troublesome synchronization issues that are difficult to debug. You can spend more of your time developing the code versus spending many hours debugging it.

Another important performance coding pattern in Java is to minimize object creation. We talked about this earlier. Creating objects in Java is expensive in terms of memory allocation and garbage collection. Use primitives when you can optimize performance, and avoid unnecessary object creation, and pass created objects to methods when possible. Put a lot of thought into this when you code, and we promise you that you will have a fast application that uses low amounts of memory. By not creating a lot of objects in frequently accessed sections of the code, you can make code run extremely fast. Then, compiling it natively with a *C2* compiler will make the code fly with performance.

One simple example where object creation hurts us with respect to performance is with a **`StringBuffer`** or **`StringBuilder`** object when we need to manipulate strings. We refer to this optimization in many places in this chapter because learning to use strings and objects properly is so important. Instead of creating a new **`StringBuffer`** object for concatenating strings in a loop, you can use the append methods to efficiently build the final string without creating intermediate objects. Consider this in your thought process and when looking at the methods available to you in objects. By reusing already instantiated objects and minimizing object creation, you can reduce memory usage and improve the overall performance of your Java code.

If you do a thread dump when an application is running, the two classes you will see the most are the **`String`** class and the **`Object`** class. If you learn nothing else, learn to be good at efficiently working with strings and objects and leveraging primitives. Nothing is more important that working with these entities efficiently. Unlike the traditional string concatenation that creates new string instances in the memory every time, **`StringBuilder`** reduces this inefficiency by allowing alterations to strings without creating new ones. This technique can significantly boost your performance when dealing with large volumes of string manipulations, as it minimizes the amount of memory consumption.

One of the slowest things you can do in Java code is use the **`+`** operator to concatenate strings. Never do this if more than one concatenation is required, trust us on this one. **`StringBuilder`** and **`StringBuffer`** are always the classes to use for this. **`StringBuffer`** is the most thread-safe approach to use when manipulating strings for multi-threaded applications. **`StringBuilder`** is not thread-safe which means it will operate a little faster for applications that do not need multi-threading safety mechanisms in place.

Beyond poor coding patterns, using too large of code abstraction frameworks, bad compilation approaches to the JVM, etcetera. one of the most detrimental things we see when it comes to poor performance is excessive logging and improper exception handling. When it comes to async logging mechanisms, remember that logging is taking up threads in the JVM. So, if you are logging in a trace or granular fashion, this will quickly consume the JVM's resources and increase your run-time costs. Generally, it translates to spinning up more JVMs to handle traffic that would not normally need to be created or vertically scaling up the JVM to a high value, which leads to poor performance in many cases. Larger heap sizes do not translate to great performance in many instances because when garbage collection happens, it is generally slower. When garbage collection happens slower, it means the application responds slower and has less throughput. It also generally translates to much more CPU utilization, which is not what we want.

The way to log in to a Java application is to log in at different levels using a configuration flag that does not require a redeployment of the code. It should be externally configurable like a switch to turn on and off. By default, in production, you should only log actionable events, unless you are flipping a debug flag at run-time for tracing/troubleshooting. The normal operation should be only actionable events in production. Meaning that when they occur you are doing something about it.

Queuing

When logging to disk, always use a **first-in, first-out** (**FIFO**) buffer. So, have the async logging thread log the message to a memory queue, which eventually goes to disk. Writing to memory is always faster than writing to disk, meaning that your async threads running in the JVM operate faster and minimize the backup to the synchronized threads that may be calling the logging mechanism. Using buffers like this for slower dependencies and leveraging queues is the key to great performance. If we have one system that operates at twenty transactions per second and another dependent system that can only operate at ten transactions per second, how do we handle this from a performance perspective? With a queue or FIFO cache where we write to memory or a persistent queue that pushes the message or request to the downstream platform. This prevents backup to the calling application and allows it to send the message asynchronously and return to the caller almost immediately. Remember though, beyond simple async logging, designing async applications is an art within itself.

Efficient exception handling

Using proper exception handling techniques can contribute to better performance in Java. Exceptions in Java can be expensive due to the stack snapshot process and the associated overhead. Java Exceptions can be optimized for better performance by following some best practices. One important aspect is to use exceptions only for exceptional situations, not for normal program flow. We cannot state this enough. You want to always have good error handling in an application and you want to only throw exceptions when an actionable error occurs. Do not create exceptions for everything! Exceptions are expensive in terms of performance because they involve creating and throwing an object, as well as unwinding the call stack to find an appropriate exception handler.

It is recommended to avoid using exceptions for control flow or as a substitute for if-else statements. Instead, use conditional statements to handle expected errors or return specific error codes or values. Yes, we know as an object-oriented programmer leveraging the Java language, this may sound like more of a functional decomposition in programming; but it is the way to assure great performance. By reducing the number of exceptions thrown in your code and logging, you can significantly improve the performance of your application or service. Not to mention reducing the lost hours for someone to transverse noisily logs chasing down red herrings which leads to long **mean time to repair** (**MTTR**).

Another way to optimize Java Exceptions is to be mindful of the exception types used. Java provides a hierarchy of exception classes and it is important to choose the appropriate exception class for each situation. Using more specific exception classes can help improve performance because it allows for more targeted exception handling. When catching exceptions, try to catch the most specific exception type possible; rather than catching a general exception type, only have general exceptions as fallbacks and not as an exception-catching pattern used uniformly throughout an application. This can help

avoid unnecessary exception handling and improve the overall performance of your code. Catching specific exceptions is much faster than catching general exceptions in java. Additionally, it is important to handle exceptions efficiently. When an exception occurs, the Java run-time system searches for an appropriate exception handler in the call stack. This process can be expensive from a performance perspective, especially if the call stack is deep.

To optimize exception handling, it is recommended to catch exceptions as close to the source of the exception as possible. This reduces the search time for an appropriate exception handler and improves the performance of your code. However, it is important to strike a balance between catching exceptions close to the source and maintaining code readability and maintainability.

Unchecked exceptions in Java have much less impact on performance versus checked exceptions. Most of the time is spent on the snapshot of the stack trace, which can be, on average, equal to executing one hundred lines of code in Java. This is not what we want! We do not just want to take large snapshots and have frequent exception calls in our application; nothing is worse for performance. We want pre-defined error codes with actions and messaging to indicate what is wrong without trying to create or read through a large stack dump.

When something fails, we want to minimize the MTTR. This means that we want to almost instantly know what is wrong when a failure occurs by providing an error code, informative message and a support document that says what to do when this error occurs. Why does the error code exist? This is key to support documentation that provides in-depth details around what the problem is, source class, action, where it is occurring basically and how to troubleshoot it, who to contact, etcetera.

Let us say that you have a web application and try to update your profile information. This means you edit some details, click a button, and the information calls a web service that writes to a database. The update profile action fails. What do we tell the user? Do we send them a stack dump, put up a *Ooops* message or *we are broken message*? No, instead, we write a user-friendly message with an error code that provides deeper meaning to the IT personnel. You want to minimize support tickets and always consider usability in everything that you develop. One approach is to popup up a message saying, we are experiencing high volume traffic, please try updating your profile later; another being, we have queued your request. Have an error number 100 as an example when it occurs and log this error message so that the IT staff can look up what it means and know what action to take, what source file to look at, etcetera. Track these errors or have alert messaging behind the scenes. This combines user friendly with information that the IT personnel need to correct the issue.

Multi-threading and asynchronous programming

Furthermore, leveraging asynchronous code and multi-threading techniques can significantly speed up Java code execution. Asynchronous programming allows you to perform non-blocking operations, enabling your code to continue executing while waiting for I/O or other time-consuming tasks in an application or web service. Java also provides important features like **`CompletableFuture`** and the executor framework to facilitate asynchronous programming. By utilizing multi-threading, you can parallelize computationally intensive tasks, taking advantage of multiple CPU cores. But as discussed when taking about immutable objects, it is crucial to always handle synchronization and concurrency issues properly to avoid race conditions and ensure thread safety and data consistency.

A **`CompletableFuture`** in Java is a class used in multithreading programming. It provides a simple way of managing and controlling the completion of work tasks in a thread. **`CompletableFuture`** will allow you to chain together tasks in a declarative manner, where the completion of one task triggers the execution of the next task. It allows you to write non-blocking code by running a task on a separate thread from the main thread and notifying you about the result of the task because it finishes executing. You can create arrays or lists of **`CompletableFuture,`** which makes it extremely powerful. It will take advantage of multiple cores of the CPU.

This exceptional feature improves the speed and efficiency of complex software applications. Before Java was good at asynchronous coding, it would have an affinity towards a particular CPU core and when you had multiple cores, it would not properly leverage them. This opened up the door for languages like Golang and RUST. But Java has risen to the challenge of balancing traffic properly across multiple cores. Here is a very basic example of a **`CompletableFeature`** that squares two numbers, but utilization of this and the executor framework will become your friend when it comes to writing fast code:

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureExample {

  public static void main(String[] args) {

  int number = 34;
  CompletableFuture<Integer> future = CompletableFuture.
supplyAsync(() -> square(num));
  future.thenAccept(result -> System.out.println("Result: " + result));
  try {
    future.get(); //getting result
    } catch (InterruptedException | ExecutionException e) {
```

```
        e.printStackTrace();
      }
    }

    private static int square(int num) {
      return num * num;
    }
}
```

Another important class is the **`ExecutorService`**. The executor Framework also plays a critical role in managing and controlling threads in a Java program. Its efficiencies become pronounced when it comes to managing thread pools. This is where its power lies. It is a crucial part of the **`java.util.concurrent`** package. It is a powerful component that enables developers to run and control tasks concurrently. This framework does a good job of simplifying thread management by providing a seamless method to execute threads asynchronously. By facilitating advanced task scheduling and load balancing, the executor framework has been able to dramatically enhance the efficiency of multithreaded programming and asynchronous operations. The executor and **`CompletableFuture`** capabilities in Java serve as the backbone of an optimized Java coding process. If you are not using them today, start!

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorFrameworkExample {

  public static void main(String[] args) {
   ExecutorService executorService = Executors.newFixedThreadPool(30);
    for (int i = 1; i <= 30; i++) {
      int taskNumber = i;
      //delegate to thread in thread pool
      executorService.submit(() -> performTask(taskNumber));
    }
  }


  private static void performTask(int taskNumber) {
    System.out.println("Task " + taskNumber + " is being performed by
" + Thread.currentThread().getName());
  }
}
```

As you can see, Java provides excellent support for concurrent execution of code through asynchronous programming and multi-threading that allows great parallel processing techniques. Asynchronous programming helps to keep the application responsive by offloading heavy tasks that are not needed immediately to be processed later. Multi-

By properly utilizing multi-threading techniques and asynchronous programming properly, you can greatly increase the speed and efficiency of your applications. We encourage you to become masters of concurrency.

In Java, without these frameworks, when talking just about basic multi-threading to run tasks in parallel, you can use the **`Thread`** class or the **`Runnable`** interface. This is the most basic way to leverage multi-threading in Java.

Using the **`Thread`** class, just define a class that extends the **`Thread`** class and then override its **`run()`** method, it is that easy!

```
class MyThread extends Thread {
 public void run() {
  // Define what the thread will do here
 }
}
```

To invoke the thread in the main class or any other class from where you want to start the thread, create an instance of your **`MyThread`** class and call the **`start()`** method.

```
MyThread mt = new MyThread(); // Create an instance of MyThread
mt.start(); // Start the thread execution
```

To use the **`Runnable`** interface, just define a class that implements the **`Runnable`** interface and override its **`run()`** method.

```
class MyRunnableExample implements Runnable {
 public void run() {
  // Define what the thread will do here
 }
}
```

Then in your application code, create a **`Thread`** object by passing an instance of your **`MyRunnable`** class to the **`Thread`** constructor. Finally, call the **`start()`** method.

```
MyRunnableExample m = new MyRunnableExample(); // Create an instance
Thread t = new Thread(m); // Create a thread
t.start(); // Start the thread execution
```

By now you should have a good feel for multithreading in Java and asynchronous programming. We encourage you to research more about the concurrency capabilities in Java leveraging these tools. Remember, when using multithreading, you need to always ensure that shared resources are properly synchronized to avoid data inconsistencies or race conditions. When choosing classes, consider which ones are thread safe and which ones you need to leverage your own synchronization. For complex applications, leverage thread pools to manage threads and try to avoid blocking in code execution as much as possible, as it can hinder the benefits of multi-threading.

Creating fast code through best practices

Efficient memory usage and minimizing CPU impact are crucial for good performance in Java. The goal should always be to use memory efficiently by cleaning objects up when they are no longer needed and not allocating humongous objects which may overload the heap when under a lot of concurrencies. We also want to minimize CPU thread time. To optimize memory usage, remember to avoid unnecessary object creation and release resources promptly. Be sure to nullify objects as soon as they are no longer needed so that the garbage collector can clean them up. This is where using finally code blocks can many times come in handy. This block of code always executes regardless of what exception is being thrown. Using appropriate data structures, the proper classes, good design patterns, and algorithms can help reduce memory and CPU resource consumption.

Minimizing CPU impact involves optimizing algorithms and avoiding unnecessary computations. One simple example is leveraging Java caching instead of recurring computation of business logic or mathematical operations that will save CPU cycles. Profiling and benchmarking early in your code can help identify memory and CPU bottlenecks. This will allow you to make the needed optimizations and improve the performance of your Java code.

Avoid unnecessary memory allocations and minimize the usage of memory-intensive operations. This can improve the overall performance of your code. Some techniques to achieve efficient memory usage include using primitive types instead of objects where possible, such as we discussed earlier around using **`StringBuilder`** or **`StringBuffer`** instead of concatenating strings with the + operator, and releasing resources promptly, such as closing streams or freeing memory. Additionally, using memory profiling tools can help identify memory leaks and optimize memory usage in your Java applications.

To start, let us talk about the speed of certain Java classes. We have talked about how using light weight classes or primitives and programming to the lowest level will almost always excel with performance over wrapper classes with high levels of abstraction. But what about some of the common classes that are used in Java. Over the years, it has been observed that some classes like **`String`** and **`ArrayList`**, depending on their usage, can be slow. As we discussed earlier, strings are one of the most frequently used objects in an application. If you profile your JVM run-time environment's heap, you will see this. **`String`** is an immutable class in Java. Thus, any modification of a **`String`** object results in a brand new **`String`** object, which can be a performance bottleneck, if extensively used. We cannot make a statement saying that **`String`** and **`ArrayList`** are slow uniformly. It comes down to understanding the benefits of each class.

The **`ArrayList`** class can be exceptionally slow when resized frequently, as it leads to array copying. It is when this happens that performance starts to get poor. Instead of leveraging the **`ArrayList`** class, consider using LinkedList**,** if you frequently add or remove elements in the middle. But regardless of what collection you decide to use, if you know the size of your list in advance, this information will help considerably; pre-size your **`ArrayList`** to avoid unnecessary resizing to achieve better benchmarks.

This is true for all collections when it comes to performance. Pre-sizing collections, arrays, lists, and other collection objects always translates to better performance by eliminating incremental allocation latencies. When it comes to whether or not to use an ArrayList or a LinkedList to transverse data for an application, the approach to take and the classes to use really depend on the application itself. When it comes to searching and accessing elements, an ArrayList is undoubtedly faster than other approaches because it provides random access to its elements with constant time complexity. However, LinkedList, including Double LinkedList, far outperforms ArrayList when it comes to adding or removing operations, especially in the middle of the list.

Java classes known for their fast performance include primitive data types such as **`int`**, **`double`**, **`float`**, and **`boolean`**. These are always fast and optimized for speed and efficiency because they directly map to the underlying hardware in many cases. Leverage primitive data types in Java whenever you can. Operations on these language types are generally faster than executing operations on their corresponding wrapper classes, such as Integer or Double, which involve additional overhead due to object creation and method calls.

The **`java.util.HashMap`** class is another Java class known for its fast performance. It provides constant-time complexity for basic operations such as get and put, making it efficient for storing and retrieving key-value pairs. Using **`HashMap`** can be problematic where large amounts of data is used. The **`java.util.HashMap`** class can exhibit slow performance when the number of elements in the map becomes too large. The **`HashMap`** uses a hash table for storing key-value pairs. But as the number of elements begins to increase, we see that collisions start occurring in many instances, leading to degraded performance.

The **`java.util.concurrent.ConcurrentHashMap`** class is designed for high concurrency and is known for its fast performance in multi-threaded environments. It is even faster than a hash table for concurrent operations. It provides thread-safe operations with minimal resource locking, allowing multiple threads to access and modify the map concurrently. This makes it suitable for scenarios where high concurrency is required, such as in multi-threaded applications or server environments.

The **`java.util.concurrent`** package overall provides classes for building highly scalable and efficient concurrent applications, and it is important to understand how to build scalable applications. Study this package well, as it is the key to high performance in Java. We talked about thread creation and the frameworks that enable concurrency and async programming. When it comes to simple threads, you can also leverage a fast thread manager called the **`ThreadPoolExecutor`** class, which allows you to create a pool of worker threads and efficiently manage their execution. By utilizing thread pools, you can achieve better performance by reusing threads versus creating new ones for each task. This holds true for connections to services, application and web servers, gateways, and databases. Any request across the network where you can maintain a pool of active connections will be faster in most instances than with the overhead of negotiation that is required for starting up something new. In many instances, when opening a new TCP/IP

connection, you may be doing a DNS lookup and SSL negotiation that can significantly degrade run-time performance.

The **`java.nio`** package, specifically the **`java.nio.ByteBuffer`** class, is known for its fast performance in handling I/O operations. It provides direct memory access and efficient buffering, making it suitable for scenarios where high-performance I/O is required, such as network communication or file handling. **`ByteBuffer`** allows for efficient reading and writing of binary data, reducing the overhead of data conversion and improving overall performance. Beyond **`ByteBuffer`**, the **`java.nio`** is another package to pay special attention to as it can solve many of the I/O problems that typically occur with web services.

Now that we talked about some of the basic fast classes let us talk about some of the common Java classes that are known for their slow performance. These classes, if not used properly, can have a significant impact on your application's efficiency and speed:

- **BigDecimal and BigInteger**: Avoid when possible. These precision classes are significantly slower than primitive types. Use them only when needed and preferably not in frequently accessed code segments.
- **Wrapper classes**: Instantiating and executing wrapped code is much slower than using primitive types directly. They also consume more memory. Use primitive types whenever possible.

One Java class known for its slow performance is the **`java.util.Date`** class. This class has several performance issues, such as the fact that it is mutable, which means that every time you modify a date object, a new object is created. Always look at classes being used in your programming and be aware of when this is occurring. It is a frequent performance killer that can lead to unnecessary memory allocation and garbage collection overhead. Additionally, to make things worse, the Date class is not thread-safe, so if you are working with multiple threads, you need to synchronize access to Date objects, which can impact performance even more. Stay away from the old data classes. It is recommended to use the newer **`java.time`** package, which provides more efficient and thread-safe date and time classes. The old Java classes around date and time are just plain slow.

The **`java.util.Vector`** class is another Java class that is slow. This class has been around in Java since its beginning. Vector is similar to an ArrayList, but it is thread-safe and synchronized. Being Thread-Safe makes it an easy class to use for multithreading. However, this synchronization always comes at a performance cost, as it introduces overhead even when working with a single thread.

If you do not require thread-safe access, it is recommended to use ArrayList or other non-synchronized collections for better performance. Be conscious of when to use thread-safe classes. Use them when executing multi-threading code, but not singular code. This should be the first thing you look for when reviewing a collection. Use the appropriate class for your application. If you need to use a LinkedList, like we discussed earlier for better performance and are leveraging multithreading or asynchronous programming, you can

consider using a **`ConcurrentLinkedQueue`**, which is a thread-safe LinkedList. There are thread-safe and non-thread-safe versions of most of the common collection classes.

The **`java.io.BufferedReader`** and **`java.io.BufferedWriter`** classes can also have slow performance in certain scenarios. These classes provide buffering for reading from and writing to files, which can improve performance by reducing the number of system calls. However, if the buffer size is too small or too large, it can negatively impact performance. It is important to choose an appropriate buffer size based on the specific use case to achieve optimal performance. Look at the file size and how many data elements are being read into the buffer as far as chunks of work. Set the buffer based on this dynamic and network packet sizing dynamics to achieve the best performance.

The **`java.util.Stack`** class is another Java class that can have slow performance. It is another one like vector that has been in Java since the beginning. Stack is a subclass of vector and inherits its synchronization overhead. It is recommended to use the **`Deque`** interface and its implementations, such as **`ArrayDeque`** or **`LinkedList`**, for stack-like behavior with better performance. **`ArrayDeque`** is not thread-safe but it is extremely fast on its own. Remember to use the synchronized reserved word if you need multi-threading abilities when thread-safe versions of collections are not available.

In Java, it is crucial to be aware of the performance implications of the classes you choose to work with. Knowledgeable choices and efficient use of these classes can help enhance the performance and speed of your application. A couple of common operations in modern IT platforms that can be extremely slow are encryption and decryption, which have a significant impact on your CPU time. Because of this we want to always make sure that we have a lot of CPU allocated to these operations when they execute and favor coarse-grained encryption/decryption over fine-grained payloads. By understanding this, you will better optimize your Java application performance.

The terms coarse-grained and fine-grained are often used to refer to the size and scope of the data that is encrypted and decrypted in this context. Here is what they mean:

- **Coarse-grained decryption**: When it comes to great software design and implementation, decrypting larger chunks of data at once should be a conscious decision for building a high-performance solution. This is one of the largest issues we see when it comes to slow applications that need to be secure. This method is generally faster and more efficient as it reduces the number of decryption calls needed, which saves CPU time.
- **Fine-grained decryption**: Decrypting with a lot of granularity significantly increases the number of decryption operations needed, consuming more CPU time and often leading to slower performance. If decryption has to happen across the network in a fine-grained fashion, you have time to go get a cup of coffee before you get a response back. This is one of the largest contributors to slowness in the platforms that we have been involved with over the years.

Though we have talked about fast and slow collections, are there any other ways that we can speed up code further? Despite their flexibility, there is an alternative that can yield better performance than large collection code bases; these are called object arrays. Yes, you read that right: object arrays, with their straightforward, low-level structure, can significantly outperform Java collections and other heavy-weight classes. This is true with all higher levels of abstraction around code. For example, if we use the URL class to bring back a web page in code, it will be much faster than using higher levels of abstraction like **`HTTPClient`** classes. The same goes for object arrays versus collections. You never want to leverage heavy weight classes or heavy weight third party Java frameworks, if you only need a small percentage of the functionality.

High levels of abstractions around code for simplification mean more method calls and slowness. It also means many more resources are taken up in the memory that the Java **`ClassLoader`** has to load when the JVM starts. Rapidly creating something in one hour versus 3 hours and then having it go into production and impact your customers for many years long term is something that we do not want to do for our frequently used journeys. Frequently used journeys in your IT platform should be lightweight and fast! Concentrate optimization efforts on the top ten journeys as a practice to balance release velocity with performance. The Java language itself provides all you need to write high performance code. If you must leverage a bulky framework, make a framework selection based on light and performant it is. Never choose frameworks that bog down your application. Shoot for programming with low-level objects, primitives, and minimal instantiation without a lot of object reuse, pooling, multi-threading, caching, etcetera. and your platform will scream with performance.

The basic structure and simplicity of object arrays mean fewer layers of abstraction and less memory overhead. Java collections such as **`ArrayList`** or **`LinkedList`** have more complex structures, involving additional objects and pointers, which leads to more memory use. They also generally take more time to load by the **`Classloader`**. For the best performance, you always want to load classes into the JVM in the order they are called. This significantly speeds up start time for the JVM. You can turn on the verbose parameter when running the JVM to see the order of objects loaded. The smaller those classes are, the faster they load!

`ArrayList` is implemented in Java as a resizable array. As more elements are added to an **`ArrayList`**, its size is increased dynamically. Each time it does this, it needs to allocate a new larger array and copy all the elements from the old array to the new one. **`LinkedList`**, on the other hand, is an implementation of a doubly linked list data structure. Each element in the linked list has two references, one pointing to the next element and another pointing to the previous element. Though not doing large array copies like **`ArrayList`**, even a linked list means more memory consumption per element compared to simple arrays. The bottom line again is always to program to the lowest level that you can for the best performance versus leveraging heavy classes where you only use a subset of the functionality.

Reusable design patterns for high performance

Object arrays or any array pose less strain on garbage collection compared to larger Java collections and data structures. An array is a continuous block of memory, while a collection is a series of disconnected object references. Recovering memory space from a collection requires more work for the garbage collector when individual objects become eligible for garbage collection, as opposed to the contiguous chunk of memory in an array. When it comes to iteration, object arrays come with an edge. Remember, arrays have a fixed, predictable structure which can be directly accessed via indices. On the other hand, the iterator pattern in Java collections introduces additional overhead, which affects performance, especially when dealing with large data sets. Object arrays, when used efficiently, can provide enhanced performance. It is important for you to consider your application's specific needs, data size, and access patterns when choosing between these two options. The lower-level object you use, the faster the **`ClassLoader`** in the JVM can load the class and the less resources that are typically used.

When it comes to Java programming, one of the areas where code can get slow is handling SQL from relational databases. Using large object to relational mapping frameworks can assure slowness and much less scale. We have seen this over and over again throughout the years. For the best performance, you want to make a direct SQL call through a JDBC driver or call a stored procedure to retrieve just the needed data in a coarse-grained way and leverage caching to minimize calls back to the database. Though many object to relational frameworks will store entity objects in memory and a full domain object graph, generally the granularity and amount of manipulation that occurs with mappings slows down an application, not to mention the large number of resources consumed. Efficient handling of SQL **`ResultSets`** is crucial for solid application performance.

Essentially, when we retrieve a data set from a database, a **`ResultSet`** object is returned. This includes multiple rows, fields, and metadata. Given their complexity, efficient processing of **`ResultSet`** Java objects with database data can significantly enhance your application's performance. Here are a few key factors to ensure great performance:

- **Limit fetch size**: Determine how many rows are retrieved at once from the database during each network roundtrip. A smaller fetch size can reduce memory consumption, while a larger fetch size can minimize network roundtrips, resulting in improved overall performance. For databases that do not have this parameter, you have to build the capability around the concept of paging in an application via a counter or the data taxonomy to break up the requests sizing.
- **Close/Nullify ResultSets**: Use the finally block in code to close **`ResultSet`** and set them to null once you are done with them, to free up system resources. Remember that finally code blocks execute regardless of an exception being thrown. Neglecting efficient cleanup can lead to a memory leak, which can negatively

impact application performance over time. Consider doing the same thing in the finally block for releasing other large objects, connections, Statement objects when they are no longer needed.

- **Overactive database cursor**: Minimize looping over `ResultSet` objects with a database cursor reaching out to the database to transverse rows. Grab the data and use it. Excessive looping over rows holds onto threads too long and blocks I/O and consumes CPU time.

Staying away from object to relational mapping patterns and frameworks is a good high-performance practice. If you want your platform to scale, stay away from mapping frameworks that promise great speed and caching with an object graph loaded into memory. You want to leverage stored procedures or a batch of SQL requests. These can eliminate the calls across the network where the application can make one call to the database, and the database can execute 20 SQL calls and come back as an example. Instead of 20 JDBC SQL calls that may incur 20-50ms of latency per call across the network combined with connection management latency, one single call can be made, eliminating the number of connections needed.

Using stored procedures or databases that support SQL grouping through a driver will significantly build scale for the IT platform. We have seen applications achieve 2000 TPS without much hardware taking this approach where solutions using object to relational mapping techniques are restricted to 100 times less scale. It makes that much of a difference when it comes to creating highly scalable solutions.

One industry design pattern that will help your platform scale significantly is called the **Transaction Script data pattern**. This is a crucial database access pattern for high performance architectures. This pattern organizes business logic by procedures where each procedure handles a single request from the presentation. Taken from *Martin Fowler's* Patterns of Enterprise Application Architecture where it originated, the Transaction Script pattern is a popular design pattern that organizes business logic by procedures where each procedure handles a single request from the presentation. Using stored procedures with this pattern will ensure that you are successful at minimizing requests across the network and bringing back data in a coarse fashion. Why is this advantageous for performance? Here are a few points:

- **Performance**: Executes database operations in a single transaction across the network, reducing the overhead of multiple transactions.
- **Simplicity**: Transaction Scripts created are simple to build and maintain. More efficient and reliable software in operation.
- **Reduced memory consumption**: Reduced memory usage compared to object to relational mapping framework approaches where an entire domain graph is maintained in memory. This frees up system resources and can have a positive effect on application speed and performance.

One of the largest impacts on CPU and performance for a distributed system is data conversion between different types of systems. If you read from a relational database and map that to an object using an object to relational mapping framework or even pull back a **`ResultSet`**, it means mapping the results to objects in memory. Then, the payload is retrieved from those objects and manipulated to a format to be returned from a service such as XML or Json; perhaps a large mapping framework like *Jackson* is used.

The data conversion and mapping, formatting, etcetera. that happens across the distributed tiers in a platform takes a lot of data and resource parsing overhead with increased CPU thread time and cost. Reformatting data payloads from the source all the way to the frontend will impact the scale and responsiveness of an application. This is why NoSQL solutions like MongoDB became popular. They can reside mostly in memory, store data as Json objects and then a browser which understands Json and JavaScript can call directly back from that NoSQL database without all of the overhead associated with mapping data to objects and different payload types across the distributed platform in different tiers.

This also allowed solutions like NodeJS to become popular for web service creation. NodeJS still screams to a halt when it has much business logic or if it were to have to parse and convert data. This is when it becomes very slow compared to other more robust solutions like Java. But if just pulling back from a NoSQL database and presenting something on the screen is needed, NodeJS is fast and an application can be created very quickly.

Let us go a little deeper into a topic we briefly touched on earlier around object to relational mapping frameworks. These involve maintaining a large object graph in memory can impact system performance considerably. It became popular years ago when **Enterprise Java Beans** (**EJB**) first came onto the scene where the concept first took hold of mapping an object to a table or set of tables in the database and having it reside in memory. Remember that every object in your graph consumes memory resources. The larger the object graph, the heavier its toll on memory. Object-to-relational mapping means much more CPU is needed to translate between the object model and the relational model. This ongoing translation can present a significant bottleneck for your application as things lazy load behind the scenes or in real time.

In a nutshell, the disadvantages are:

- **Too much memory**: Entity objects can consume a lot of memory. Each object graph holds references to other objects and other attributes/methods, etcetera. which impact the system's memory and run-time. This slows down the IT platform's performance.
- **Too much CPU**: Object-to-relational mapping takes a lot of CPU resources. This impacts performance and scale.

If, for some reason, you are forced to use an object to relational mapping framework, simplify your database model, thus simplifying your object graph and minimizing relationships. Leverage caching and minimize the number of database hits wherever

possible. In line with the benefits of the Transaction Scripts noted earlier, adopting practices that simplify code while maintaining efficient database interactions will lead to leaner, memory-efficient applications. Therefore, it is important to optimize your entity models and object-relational mapping techniques to improve your Java application performance.

Whether you chose a high-performance solution like the Transaction Script approach or a slower object to relational mapping one, one important strategy to expedite your Java application and improve performance is by adopting caching in general. Beyond performance consider using caching when you can to not require calls downstream to build in more resiliency. Essentially, instead of re-fetching the original data or recomputing results, the application will return a copy from the cache. Thus, caching substantially minimizes response time while optimizing system performance.

In a distributed platform, you need to decide to cache in an enterprise level cache or global cache. Enterprise level caches are where requests have to be made across the network to a central cache, incurring network latency. Application-level caches will make the application run faster without the overhead but consume resources in the JVM you are running your application in. As a rule, store small sets of static data that do not change very often at the application layer at a minimum. Then, if the data changes, if you have a horizontal set of pods or application servers, send a pub/sub event when the data element is changed asynchronously to update the other Pods. If it is a small set of data (less than 100 Mbytes), then load the cache from the database as the pod or application server starts. The more you read from cache, the less requests go across the network, the faster the performance and the greater the capacity of the platform. So, to reiterate the advantages:

- **Performance**: Caching improves performance and scale by reducing the calls across the network and by providing rapid access to frequently used data.
- **Reduce load on downstream systems**: Caching data reduces the load on downstream database resources and services.
- **Reliability**: Isolate data from downstream dependencies and failures, making the platform more resilient and reliable.

Java offers different caching strategies that could be implemented based on your application requirements.

- **Least recently used (LRU)**: This algorithm discards the least recently used items first when the cache is full. It assumes that items accessed recently will likely be used again in the near future.
- **First-in first-out (FIFO):** This caching algorithm removes the earliest data first when the cache is full. Popular when you have a slow downstream system or component where you need to queue.
- **Least frequently used (LFU):** LFU eliminates the least frequently used data first.
- **Most recently used (MRU):** This caching algorithm operates on a theory opposite

- **Random Replacement (RR):** This method randomly selects a candidate item and discards it to make space for the new data.

Here is a straightforward Java caching scenario using the **`ConcurrentHashMap`** class:

```
import java.util.concurrent.*;

public class BasicCacheExample {
 private final ConcurrentHashMap cache = new ConcurrentHashMap<>();

 public Object getFromCache(String key) {
  return cache.get(key);
 }

 public void addToCache(String key, Object value) {
  cache.put(key, value);
 }

 public void removeFromCache(String key) {
  cache.remove(key);
 }
}
```

To conclude, as we have mentioned in other chapters, caching is a handy strategy when you need to increment performance, deal with large traffic loads, or provide better user experiences. However, developers should carefully consider their caching strategy, keeping in mind that an inadequate caching scheme may lead to stale or inaccurate data. There are multiple classes and libraries that can be used for caching, such as Memcache, EHcache, or distributed caches like Redis, but the principle is the same. The differences are just related to algorithms around how to keep caches up to date or if they are remote, along with how to expel objects when the data is no longer valid.

One poor man's cache against the database is to use something like the above, where the key is the SQL, and the value is the ResultSet. Then, when the database is updated from another process, trigger a removal of the SQL key. As a practice, you pull from the cache first, and if the object is not there, go to the database and retrieve the information. Caching can significantly enhance the speed and performance of your Java applications. Whether you use Memcache, EHcache, distributed caches like Redis, or even a poor man's cache with SQL as the key and ResultSet as the value, the key is to ensure effective and consistent cache management. Always remember, if you hit a snag with your performance, you should first pull from the cache. If the required object is not there, only then proceed to the database. When a write operation occurs against the database, you need to receive a pub/sub event or other communication event that tells you to refresh that element.

Beyond patterns around caching, coding for performance means coding good design and performance patterns. The most famous book on object-oriented design patterns is called *Design Patterns: Elements of Reusable Object-Oriented Software*. We would encourage you to have a copy of this book and keep it forever, it is truly timeless. This famous book was written by the top experts in software development at the time of writing decades ago:

Erich Gamma, *Richard Helm*, *Ralph Johnson*, and *John Vlissides*. It is referred to as the **Gang of Four** (**GOF**).

Most books today seen on design patterns are derivatives of what is presented in this popular book. The patterns are very relevant and used frequently in all IT platforms.

These are the most important design patterns independent of the object-oriented computer language being used. They consist of 23 patterns that are classified as creational, structural, and behavioral. The creational patterns, according to the GOF are as follows:

- **Singleton pattern**: Ensures a class has only one instance and provides a global point of access to it.
- **Factory method pattern**: Defines an interface for creating an object, but leaves the choice of its type to the subclasses, creating instances of the appropriate subclass.
- **Abstract factory pattern**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder pattern**: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- **Prototype pattern**: Creates new objects by copying an existing object, known as the prototype.

The structural patterns are as follows:

- **Adapter pattern**: Allows the interface of an existing class to be used as another interface.
- **Bridge pattern**: Separates an object's abstraction from its implementation so that the two can vary independently.
- **Composite pattern**: Composes objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.
- **Decorator pattern**: Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Facade pattern**: Provides a unified interface to a set of interfaces in a subsystem, making it easier to use.
- **Flyweight pattern**: Minimizes memory usage or computational expenses by sharing as much as possible with related objects.
- **Proxy pattern**: Provides a surrogate or placeholder for another object to control access to it.

The behavior patterns are as follows:

- **Chain of responsibility pattern**: Passes a request along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- **Command pattern**: Encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of the parameters.
- **Interpreter pattern**: Defines a grammar for interpreting the sentences in a language and provides an interpreter to deal with the sentences.
- **Iterator pattern**: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator pattern**: Defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly.
- **Memento pattern**: Captures and externalizes an object's internal state so that it can be restored to this state later.
- **Observer pattern**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State pattern**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy pattern**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template method pattern**: Defines the skeleton of an algorithm in the superclass but will let subclasses override specific steps of the algorithm without changing its structure.
- **Visitor pattern**: Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Most abstraction frameworks and classes follow these patterns under the covers. Following these patterns in day-to-day coding will ensure that minimal code is used and that the code is fast and easy to maintain. Another book that builds upon GOF for patterns is *Enterprise Integration Patterns* by *Gregor Hohpe* and *Bobby Woolf*. The patterns in this book that are used in most distributed cloud platforms today are as follows:

- **Message channel pattern**: channel that connects producers and consumers.
- **Message pattern**: unit of data exchanged between applications.

- **Command message**: specialization of the message pattern where the message carries a command to the recipient to perform.
- **Document message**: Specialization of message pattern where the message is self-contained and well defined.
- **Event message:** Specialization of message pattern where the message represents an event.
- **Request-reply pattern:** messaging pattern where the requestor sends a message and waits for a reply.
- **Point-to-Point channel:** connects a single sender to a single receiver.
- **Publish-subscribe channel:** Broadcasts messages from sender to multiple receivers.
- **Datatype channel:** Channel that transfers only messages of a specific data type.
- **Invalid message channel**: Handles messages that could not be processed.
- **Message router**: Routes a message from one channel to another based on conditions.
- **Content-based router**: Routes messages based on content of the message.
- **Dynamic router**: Routes a message to one of a number of alternative endpoints.
- **Recipient list**: Routes a cop of a message to multiple recipients.
- **Message translator**: Coverts the format of a message from one to another.
- **Envelope wrapper**: Wraps a message with additional information.
- **Content enricher**: Enhances a message by adding additional content.
- **Message filter**: Filters messages based on certain criteria.
- **Message history**: Records information about the history of a message.
- **Message store**: Stores messages for future retrieval.
- **Smart proxy**: Proxy that provides additional functionality such as monitoring or logging.
- **Messaging gateway**: Encapsulates messaging specific code and provides simple interface to the application.
- **Messaging adapter**: Allows an application to interact with a messaging system using a standard interface.

Beyond these basic object-oriented coding and messaging patterns, *Patterns of Enterprise Application Architecture* was written back on 2002 by *Martin Fowler* is one of the best industry books out there for large scale distributed platform design. We have followed his work for many years. The design patterns listed go deeper into distributed platforms beyond GOF object-oriented coding patterns. The patterns consist of the following:

- **Domain model**: Problem domain as a set of interconnected objects
- **Table model**: Domain logic with one class per table where each instance of a class represents a record in the database
- **Data mapper**: Separates in-memory objects from the database
- **Identify map**: Maps objects and looks up objects as they are requested
- **Unit of work**: Maintains a list of objects affected by a business transaction
- **Gateway**: Simple interface to complex system
- **Service layer**: Application boundary and set of external services available
- **Value object**: Small object that represents a simple entity
- **Composite view**: Aggregates multiple views
- **Front controller:** Centralizes control logic versus duplicating it
- **Template view:** Abstract view of an existing view
- **Two-step view:** Divides view into two parts for viewing versus editing
- **Application controller:** Manages flow of control for an application
- **DTO:** An object that carries data between processes
- **Session state:** maintains conversational state between requests
- **Page controller:** handles request and determines appropriate view to render
- **Lazy load:** Delays the loading of an object until it is needed
- **Repository:** Mediates between domain and data mapping layers
- **Active record:** Encapsulates data access within the domain object
- **Special case (Null object)**: class that provides an interface that clients expect but with no operation

Since writing the book in 2002, *Martin Fowler* has incorporated multiple modern principles around designing fast and flexible software including:

- **Micro-services**: architecture collection small independent services each with its own business logic and data storage
- **Saga pattern**: pattern for managing distributed transactions across multiple micro-services
- **Event sourcing**: pattern where state is determined by sequence of events
- **Command query responsibility segregation (CQRS)**: command and query responsibilities are separated into distinct components

- **Database sharding**: pattern for horizontally partitioning a database to improve scalability
- **Event-driven architecture**: architecture where components communicate with each other through events
- **Remote procedure invocation**: pattern for remote invocation of procedures
- **Integration databases**: pattern where an integrated database is used to simplify data across multiple services
- **CI/CD**: practices to ensure that code is always in a deployable state

These are the best patterns out there for designing great platforms when looking at the basics. They are the basis for what other books have derived their work. We recommend you look closer at the design patterns above and do research into these books and the information that is portrayed there.

Another source of understanding distributed platform capacity and performance we cannot overlook is from a book written by *Neil Gunther* many years ago; well before he came up with the USL. It is not a well-known book and one that can be hard to locate. However, it is one that is crucial to understanding the performance of distributed platforms. It is called the *Practical Performance Analyst* by *Neil Gunther*. This book will provide you with all of the formulas for queuing patterns to understand how upstream and downstream platforms can impact one another from a performance and capacity perspective. It will help you understand the concepts of performance in distributed platforms. This book does an excellent job at covering queuing theory. Though it is an old book, it also is one that is timeless like the others.

Conclusion

In conclusion, we went deeper into the JVM optimization and code optimization and tied these two dynamics together. This chapter provided you with a wealth of information on how to write fast code, including techniques for multi-threading and asynchronous programming. Industry design patterns were discussed, and sources to obtain additional information were provided. We talked about fast classes, slow classes, objects that created new instances of themselves in a loop, and how thread safety impacted response times. Caching was again discussed since it is so important to performance.

In the next chapter, we are going to go deeper into database-related optimizations for distributed platforms. These last two chapters went pretty deep. Stay tuned as we continue to go deeper into understanding how to make IT platforms fast!

Chapter 9
Database Optimization

Introduction

Most IT platforms today need to persist data, and they use a database to do this. Optimization comes into play when we talk about the process of improving the speed or efficiency of a database system. The goal is always to improve the user experience and have great responsiveness at the lowest cost. Whether the database is a common structure across web services or a separate one per web service, such as with micro-services, the data being **CRUD** must be fast, or the IT platform quickly comes to a halt.

When the proper keys are not leveraged, full table scans happen, or when too many temporary tables and joins occur against large-volume databases, we can clearly see performance degradation. The volume of data in tables impacts performance. Also, the structure of the database, including data relationships, influences transactional speed for an IT platform as well as the query, primarily how well it is optimized.

There are a lot of considerations around the normalization of data, data redundancy, and dependency, in addition to how the data is stored on disk and in memory. You need to understand entities and relationships and data integrity and security and privacy as well to consider how encryption or decryption may be a factor. How the data is accessed plays a great influence on how well the IT platform performs as well.

Understanding database optimization and implementing strategies for improvement to reduce runtime costs and improve responsiveness and scale can make a dramatic difference to your customers. In this chapter, we will continue to talk more about how the data access patterns and join that with database optimization techniques. This will tie together the important concepts around data access.

Structure

This chapter covers the following topics:

- RDMS optimization basics
- Stored procedures
- NoSQL databases

Objectives

In the last chapter, we talked about some of the data access patterns for database access, such as Transaction Script or implementing a domain model with entity objects. In this chapter, we are going to go deeper into the database and talk about optimization and how to make things faster. We will discuss different types and different tips and techniques to improve performance. Optimization includes augmenting queries and tuning the database to run faster with less resources to provide the same functionality as before the optimization took place.

You must have a basic understanding of databases to benefit from the information presented in this chapter. It is important to have a good grasp of how to work with different SQL statements and data types as a precursor to reading further. Let us get started!

RDBMS optimization basics

Optimizing your database is like tuning Java code. It is an essential process that enhances performance and efficiency, leading to better user experiences and cost-effective IT platforms and better data management. It does not matter if you are working with a **relational database management system** (**RDBMS**) or a NoSQL database. An optimized system will make a world of difference to your operations. Let us go further into how these optimizations work starting with the databases most people are familiar with, an RDBMS. Once we have a better understanding of relational databases that leverage SQL, we will compare the benefits of relational database platforms versus NoSQL platforms. We will provide some handy and simple tips to enhance the speed of your database.

Most common performance issues

Indexing, using the proper keys, and referential integrity play an important role when it comes to fast access. This is where most problems

establishing the proper keys or indexes when a query happens, a full table scan occurs when looking for the data versus having an index that takes you directly to the information you need quickly without searching all of the other rows in a table. Think of an index, the appendix in the back of a book, which takes you to the exact paragraph or page that you are looking for. Indexes greatly boost the speed and efficiency of data retrieval, and if you have slow SQL performance when it executes, missing indexes are always the first thing to look for. Creating keys and indexes needs to be a part of creating your IT Platform and these should be aligned with every single query that is executed. As queries are created, look at how the database is accessed and establish the needed indexes. If you are retrieving a name from a database in your SQL statement or have a where clause that has a condition on a field, then these fields should be indexed.

Keys

Keys, are a little different but just as important. These are unique identifiers that identify a particular record or row in the database. They do not have to be a single value; they can be a combination key of multiple values to achieve uniqueness for the record or records. There are primary keys for a table and generally foreign keys which correlate with primary keys in other tables, and when joining tables, this is how you isolate specific information and helps you achieve referential integrity. If you delete a record in one table, the statement should be written to join the other tables, etc. Referential integrity ensures that relationships among different data sets remain consistent and not full of errors.

Basic SQL optimization

When it comes to relational databases, referential integrity is a significant concept in databases leveraged for IT platforms. It enforces the logical consistency of data across tables. It involves using foreign keys that reference the primary key of another table. If you leverage a database without referential integrity, you may accidentally insert invalid data into your tables. This leads to erroneous results, which can be difficult and costly to fix and impact the quality of the platform overall.

Imagine a scenario where you need to fetch an employee from a large database, with each record corresponding to a single person. If you do not optimize your query, you might simply write something like:

```
SELECT * FROM Profile
```

This would bring you back all of the employees in the table and all columns and then you could search through the list with the software to find the record you needed. However, it takes a lot of time to retrieve a list of all employees, and imagine if that list has 50,000 people in it. This simple query will pull every single row from the Employees table, including all of the columns each row has. This is not only slow but also taxing resources on the database, especially if it is a large table with a great deal of records.

In order to speed up data access and reduce the load on the database, it is important to be specific in queries. If you only need one employee's name and contact details, you want to specify this in your query.

```
SELECT name, address, email, phone FROM Profile
```

This query will only return 4 columns instead of getting all columns. So, if we have a total of 20 columns in our table, we reduce the information to only 4 reducing the information retrieved considerably. This enhances the system's performance. You want to always follow the concept of data minimization, where you only gather what is necessary. This minimizes resources being used by the database and assures great performance and efficiency.

Another effective technique is using filters to reduce the amount of data processed using the WHERE clause. Let us say you have a database of many companies and setup profiles of employees for each company. This database may have millions of rows in it, and you want to just search for names for a single employer, speeding up access.

```
SELECT name, address, email, phone from Profile
where organization=" IBM"
```

Efficient SQL optimization comes down to being more specific in your queries and filtering the needed data, isolating a group of data to be accessed quickly. It means querying using keys and indexes to get to information quickly.

Incorrect or missing indexes and keys will lead to full table scans trying to find information like a needle in a haystack, leading to longer response times and reducing the overall efficiency of the database. Every single statement to the database should be reviewed and setup with the proper indexes and keys. But let us consider the caveat here. Though read operations in a database will scream with performance, write operations are typically the slower commands to the database and it can be slower based on the number of indexes that you have.

There are a few things you can do when it comes to inserting or updating data in the database to make it faster for the customer or service to insert the data. One approach is to batch process SQL commands to reduce I/O operations against disks or use an unindexed staging table for real-time transactions where you have a batch process or even a field trigger process to move the data from the staging table to indexed tables, supporting referential integrity. So instead of inserting one record by another group multiple insert statements and batch execute. Another popular approach is write-behind caching or FIFO caching. This method temporarily stores write operations in a high-speed cache and executes them later.

When it comes to SQL optimization, it is important to identify the columns that are frequently used in the **`where`** clause or **`join`** conditions and create indexes on those columns. Regularly analyzing and optimizing indexes based on the query workload can greatly improve query performance. Another important practice is to minimize the use of wildcard characters, such as **`%`**, at the beginning of the search pattern in the **`where`** clause.

When a wildcard is used at the beginning, it forces the database engine to perform a full table scan, which can be slow for large tables. Instead, structure the query in a way that allows the use of indexes as much as possible and use the wildcard at the end of the search pattern.

When optimizing your RDBMS SQL statements, consider joining tables where logical relationships exist. Joins are generally fast and can often remove the necessity for multiple separate queries. However, joins must be used with caution. Poorly constructed joins can result in cross-joining and return far more data than expected, which may slow down the system. De-normalizing data can mean consuming more resources, but it can also mean much faster performance for the calling application with less joins. For most organizations, databases form the backbone of their information systems. However, to fully capitalize on available data, it is crucial to ensure that your databases are well-optimized for performance.

In SQL statements, always use specific search criteria and avoid unnecessary OR statements, which will help any database optimization solution baked into an RDMBS choose the most efficient execution plan for the query. Remember earlier; we said that one of the most impactful things to performance is joining too many large volume tables. To optimize queries, you want to avoid any unnecessary joins. Each join operation can impact query performance, where more volume is determined by how much in many instances. It is important to eliminate unnecessary joins and only include the required ones using the proper join types, such as left join or inner join. A **left join** is when two tables are combined from a data perspective, and the query returns all records from the left table that match the records in the right one. An **inner join** joins records from both tables where the criteria are met.

Understanding the data model and the relationships between tables is crucial for optimizing join operations, so when creating queries and setting up indexes and keys, pay attention to the relationships and how an application or applications will access the data. When it comes to achieving great performance when retrieving data for an IT platform, caching query results can be an effective optimization technique. For queries that are executed frequently, where the result set does not change very often, caching the result can significantly reduce the database load and improve response times. The best way to do this is by implementing a caching layer in the application or utilizing database, including features such as materialized views or query result caching.

Optimizing SQL queries involves understanding and analyzing execution plans generated by the database optimizer. What are execution plans? Execution plans are what most RDBMS platforms use to provide insights into how the database engine intends to execute a query and can help identify potential performance bottlenecks. Essentially, they provide the steps determined by the query optimizer for the database and detail how specific data is retrieved, such as sorting, aggregation, and paths to data. For databases that leverage execution plans, understanding them comes in handing for optimizing a database for optimal performance.

Databases like *Oracle* have a great deal of reporting on database performance with **Automatic Workload Repository** (**AWR**) reports. This is a powerful tool embedded in *Oracle databases* that collects performance statistics for problem detection and database optimization. Other database solutions have similar concepts and choosing solutions that will do performance analysis is a good idea. When choosing a database, leveraging one that provides performance statistics and insights into the database internals will ensure that you have the flexibility you need to build fast IT platforms.

SQL execution plans come in handy in diagnosing issues and tuning your database for optimal performance. They will identify where you need to optimize, where you have missing indexes, inefficient joins or too much volume in data being read. This should be reviewed often.

For fast performance in the database, you want to enable any query optimizers available to find efficient query plans that will save CPU and I/O operations and reduce the number of disks, reads and writes, and frequency. For databases that support pre-compiling, you want to leverage prepared statements. This is an important feature where the database pre-compiles SQL code and stores the results, separating it from data. It usually does this for the first request, and subsequent requests are much faster, not needing to pre-compile again.

When using prepared statements, the SQL query is parsed, analyzed, and compiled and will perform optimizations by choosing the most efficient algorithm leveraging relevant keys or indexes. When using these, the statement binds query parameters, and the database system executes the query. This can be repeated multiple times quickly with different parameters each time if the same query is being run repeatedly.

Normalization versus denormalization

When it comes to performance another optimization technique relates to the concept of normalization and denormalization. Normalizing your data involves breaking it down into smaller tables with unique keys to minimize redundancy of storage in the database and use fewer resources; it contributes to referential integrity. Databases that have many different applications accessing the data generally are highly normalized. Normalized databases are common with an RDBMS and are the best option when multiple applications are accessing it.

Denormalized databases reduce the number of tables, store more redundancy in tables and take up more database resources. So, instead of joining 20 tables, you can de-normalize the data into fewer indexed tables that are catered exactly towards the application making the request and the request mapping for the application. If you do this, you find that, though the database itself is storing more information, the access to that information ends up being faster for an application because the data is catered towards the request being made and the mapping is faster.

The largest latencies for applications are joining a bunch of tables with large volumes of data and mapping that data to java objects. If we have 5 tables representing a user profile in an IT platform, instead of joining on those 5 keys, we have one table with a combination key to represent individual records. Then, we make a request with the combination key and map the results to a single object versus five separate objects. We can reflect back to the discussion around the Transaction Script pattern or patterns where, instead of storing a domain object graph in memory with large datasets and relationships representing different tables, we just store a single object with the results for the specific query.

Stored procedures

Another one of the most important tools for the performance of an IT platform is **stored procedures**. Stored procedures offer huge performance and scale benefits when it comes to database optimization and an IT platform's scale. They are pre-compiled SQL statements stored in a database that can be invoked from Java or directly from the database. Stored procedures, unlike regular SQL statements, significantly improve database performance. Remember back in chapter 8 when we talked about how native code compilation for Java had a large impact on performance? This is the same situation but for a stored procedure. Stored procedures are compiled once and stored in executable form, so the execution time is far quicker. Beyond performance, you also get the advantage of better or aggregated security, where the procedure acts as an abstraction layer on top of the tables blow it.

Stored procedures can bundle a number of SQL statements into a single procedure and reduce network congestion and latency for requests. Instead of making 20 requests across the network to retrieve data with individual SQL calls, you can make one single call and execute the 20 SQL statements needed. They reduce server and network loads significantly. If you are building a platform that needs to scale, we recommend that you leverage stored procedures heavily. The only caveat is always keeping business logic out of the database for proper separation of concerns is an important practice. So, the benefits that you get are:

- **Traffic reduction**: The procedure is saved on the server, natively compiled, and executed there; there is a decreased necessity for SQL commands to be sent over the network, reducing network traffic and accelerating execution times for overall data retrieval. Remember that fewer connections, requests, and more database efficiency lend themselves to more scale at lower cost.
- **Reusable**: These procedures can be used by multiple applications, web services, and other users. So, multiple applications or web services do not need to do the optimization work or heavy SQL optimization efforts.
- **Efficient**: A stored procedure allows grouping of SQL statements into a single procedure, streamlining the construction and management of complex operations

We have talked a great deal about an RDBMS. Which RDBMS do you leverage? A good place to understand what is being used in the industry is to go to the following website:

DB-Engines Ranking: **https://db-engines.com/en/**

As of 2024, *Oracle* was the most popular RDBMS database, followed by MySQL, SQL Server, PostgreSQL, and DB2 in order of usage. While relational databases can appear rigid due to their fixed schemas, they have come a long way in pushing themselves to be more distributed and multi-model. But an RDBMS is not the only option and works best as an RDBMS; when operating in another model, they are not impressive. This is where NoSQL databases excel.

They provide unparalleled flexibility in dealing with different types of data structures. Many of these operate entirely in memory with smaller footprints than RDBMS and scream with speed at the expense of little referential integrity. As of 2024, the most popular key/value store databases were Redis, DynamoDB, Cosmos DB, Memcached etcetera.

For Document stores, in order of popularity, MongoDB, DynamoDB, Databricks, Cosmos, and Couchbase. For time-series databases, which are great for storing performance metrics or any data over time, in 2024, the most popular databases were InfluxDB, Prometheus, Kdb, Graphite, and TimescaleDB. Finally, we come to graph databases, which are an impressive concept for data storage. The most popular databases in 2024 for graph databases were Neo4j, Cosmos, Aerospike, Virtuoso, and ArangoDB.

NoSQL databases

But to really make a NoSQL database fast, you need to design your application to accommodate the database's unique features. This involves creating a data model that fits your access patterns, similar to what we talked about with de-normalized data for an RDBMS. Understanding the importance of the key-value store in retrieving data is vital. Partitioning keys to distribute data evenly across nodes can result in significant performance improvements. Regular maintenance tasks for any database RDBMS or NoSQL, such as data compaction and repair, will further contribute to the efficiency and performance of databases.

Let us look at another crucial aspect of database optimization when it comes to cloud software, which is scaling. Just like when we talked about micro-services and scaling in Kubernetes, the database is no different. Scaling is about expanding your database's capacity to handle more load. It can be achieved either horizontally or vertically. This is true for both relational databases and NoSQL databases. The approach might be different depending on the platform and its specific characteristics. The scaling types are the same as with Kubernetes or any cloud solution:

- **Horizontal scaling**: Add servers to the network to share the workload. Boost capacity by adding more machines horizontally in the database cluster.
- **Vertical scaling**: Add more power (CPU, memory, etc.) to an existing machine.

What scaling solution should we leverage? In most situations, for real-time cloud applications, we see horizontal scaling used. When workloads are not consistent, or traffic

consistent workloads that provide high computing power, like batch operations. Ideally, in any IT platform you want to separate out your batch and real-time infrastructure when you build out your architecture. You do not want batch processes impacting real-time processes, so the more isolation, the better. Having a machine-run batch or large compute applications with vertical scaling versus horizontal scaling is a great option.

Now, let us discuss another important concept when it comes to database optimization with large datasets, database sharding. This technique is all about splitting and storing data across multiple databases or servers to enhance performance and scale in the cloud. It significantly enhances performance and balances where data is stored, making it efficient to retrieve. Let us talk about the benefits and how to implement sharding in both and RDBMS and NoSQL databases:

- **Scalability**: Horizontal partitioning to scale easier as the database grows.
- **Performance**: By dividing the data load up, results from database queries more quickly.

With an RDBMS, sharding can be done but can be more challenging due to referential integrity dependencies. It is important to understand the data well. First, you identify the key by which to shard (could be a user ID, location, etc.). Then, you create the shards, which are essentially separate databases based on the key. Make sure that you have a scheme to maintain referential integrity across the shards and then modify the application to connect to the shard needed based on a key.

It is much easier with NoSQL. Most NoSQL databases support sharding natively and you do it in a similar fashion. To leverage sharding in NoSQL databases, you divide the data into chunks and distribute them across multiple shards. You want to spread data or the load evenly for the most scale, availability, and performance.

Let us talk about NoSQL more and consider the different database types as a starting point. These are as follows:

- Document-oriented
- Key-value stores
- Column-based stores
- Graph databases
- Time-series databases

These different types of NoSQL databases each bring their own set of benefits and disadvantages when it comes to performance. Understanding these can help dictate which NoSQL platform best meets your needs. One of the databases that we have seen many large organizations use is MongoDB, which is a document database. This is a great solution that scales well, is robust, and leverages memory very well.

The largest advantage of using MongoDB versus an RDBMS like *Oracle* is that it eliminates

The browser's language is typically JavaScript, which can easily read a document type of Json. If the database is storing the data in Json format, the web service in the middle between the browser and the database needs to do minimal, if any, parsing, which significantly speeds up an application. When you do less parsing and data mapping, you leverage less CPU and memory resources. If leveraging a database like MongoDB, which can store a great deal of that Json in memory, your platform can instantly become extremely fast. Solutions like NodeJS like to leverage document databases like MongoDB for speed. Document-oriented databases offer:

- **Performance**: With its ability to handle huge volumes of Json records, a document-oriented database optimizes read/write performance. It also reduces mapping/parsing across layers for web applications, significantly speeding up IT platforms.
- **Schema**: Document-oriented databases do not require a predefined schema, which makes them flexible with respect to the information structure being stored. This is an important characteristic because it provides adaptability and supports fast changes.
- **Scalability**: Easy to scale and lighter weight than bloated feature rich RDBMS platforms in most cases.
- **Replication**: Most NoSQL document databases support simple replication, meaning that they create multiple copies of the data. This ensures data safety and enhances accessibility and performance.

Another popular database type is key/value pair databases. These have the following attributes:

- **Very fast read/Write speed**: Simple key lookup like a HashMap in Java. This eliminates the need for complex query processing.
- **Scalability**: Easy to scale and partition data across multiple servers and maintain high performance and availability.
- **Flexibility**: These databases can accommodate a wide variety of data types. They are suitable for many different application types.
- **Concurrency**: Handle a great deal of concurrency with great performance.

Another type of database that is extremely performant for analytics or query-intensive operations is the column-based database. These databases have improved data compression, great query performance, minimized disk reads, and flexible indexing. When it comes to performance, it is determined by how the data is stored and accessed. Using columns, the database will compress exceptionally well and fast; this reduces disk space requirements and reduces I/O time. The data is also only queried for a column that is indexed versus the entire row, which means fast data retrieval. Column databases bring a lot of value when it comes to needed fast performance and analytical insights around data.

A discussion about NoSQL database types would not be complete without talking about a graph database. Graph databases are used when relationships are complex and unpredictable. They are object graphs in memory in many cases and have great performance. They are perfect solutions for real-time analytics, social media networks, or applications that may have to transverse complex relationships. These are very adaptable and can retrieve complex data in a single operation extremely fast.

Last but not least, Time series databases are specialized databases designed with the explicit goal of handling data change over time. They are equipped with a variety of advantages that aid in achieving efficient performance. We see these used heavily with monitoring, performance, and large dataset analysis over time. Here are some major performance aspects:

- **Time oriented**: Data is inherently time-oriented. This means that time-based queries are processed much faster than other databases. It works well for tracking things like web traffic or performance over time, monitoring, etc.
- **Compression**: Time series databases are adept at condensing data stored over time.
- **Fast writes**: Unlike many databases where writes are slow, Time series databases have been built to be fast. They can inject a massive amount of data rapidly and efficiently.
- **Scalable**: Easy to scale to accommodate large amounts of data with great performance.
- **Time series functions built-ins**: These databases have a lot of functions that are dedicated to time series analysis for large datasets.

Time series databases are great for real-time monitoring, in-depth trend analysis, and efficient data management. Put a Grafana or other dashboarding solution on top of one of these and you can really get some meaningful insights from raw data.

As we discussed earlier, many times, the database chosen is based on not only its core capabilities but also how it impacts the platform with respect to transformation latency. One of the slowest operations in an IT platform is data mapping and parsing data from one format to another as data moves from one tier of a distributed platform to another. Consider this when choosing a database solution: look at your platform as a whole. The less transformation of data, the better performance and scale.

An RDBMS has very structured data, which means that it will need to be parsed and mapped to objects, which will adversely impact the performance of the platform when dealing with large volumes of data across a distributed architecture. NoSQL databases, on the other hand, do not need any parsing or object mapping in many situations. It is crucial to understand that choosing the right database solution for the right use case is as important as factoring in the IT platform as a whole in the decision-making process. As far as relational or NoSQL solutions, it is important to consider if you need data consistency

Conclusion

In conclusion, in this chapter, we have talked about some simple things we can do to achieve good performance for the database itself and retrieve data from the database. Indexes, keys, sharding, and horizontal and vertical scaling were all discussed, as well as the advantages of different database types. Transformational latencies from the database across a multi-tiered architecture, as discussed, are something to pay close attention to with respect to a platform's performance. We hope that you learned something in this chapter or that it was a good refresher about database performance and simple things that can be done to achieve minimal latency and maximum scale.

In the next chapter, we will talk about architecture along with some case studies and design patterns for high-performance cloud architectures.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 10
High Performance Architectures

Introduction

This chapter of the book will talk about architecture and some case studies. It introduces you to some of the challenges and realities around trying to achieve good performance for large distributed applications that operate in the cloud. When it comes to building high-performance cloud architectures, no doubt there is a lot to consider. Whether you are a CTO, an architect, a performance engineer, or a developer, there is a world of considerations to keep in mind. This chapter aims to make it all more manageable. We will talk about specific case studies and established design patterns for sculpting effective, efficient cloud architectures in this chapter.

When the cloud came into view in the industry, it changed the way we think about distributed architectures, scale and self-healing. It also changed the way we think about data storage and access. Building a high-performance cloud architecture is more than just about innovation; it is important for your business. It can improve operations, increase customer satisfaction, and open avenues for new revenue when architectures have the needed flexibility. These new architectures that have evolved in recent years have enhanced speed, resilience, flexibility, and scalability inherent in modern cloud solutions. Most importantly, they allow flexibility to create fast products in a much shorter window than ever before.

Structure

This chapter covers the following topics:

- Introduction to software architecture
- Separating batch and real time
- Pervasive design patterns
- Improving API performance

Objectives

In the last chapter, we talked a great deal about the database and how to optimize database transactions. Now that we have come so far into discussing good design patterns for performance for the web and database, it is time to talk more about basic architecture and how it impacts performance. We will discuss different architecture types and then talk more about what companies are doing around software creation and performance. You must first have a basic understanding of software development practices as a precursor to reading this chapter on architecture to benefit from the information presented.

Introduction to software architecture

When we talk about software architecture, we are talking about the structure of the IT platform. It ultimately represents how different architecture layers and components interact with one another. There have been a lot of different architecture types in the last 30 years that have evolved, and lessons have been learned from each of them.

When it comes to performance, newer architecture does not necessarily mean that the architecture will be faster. When choosing an architecture, considerations should not only reflect the flexibility of the system from a design perspective but also balance performance and runtime costs. This may not mean that you use the latest and greatest architecture; always use the one that balances what you need. Never just look at build costs; look at runtime costs; this is where performance comes into play. Let us delve into the different types of software architectures to better grasp this intricate subject.

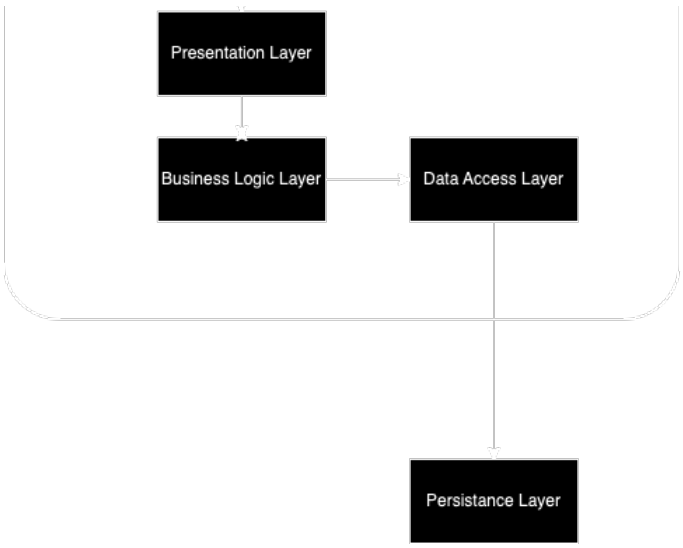


Figure 10.1: *Monolithic architecture*

Monolithic architecture

First, you will hear the term **monolithic architecture**, which refers to a system where components are generally tightly coupled with one another in the architecture. The tight coupling makes this architecture simple to develop, test, and deploy. However, the interdependency of components can also lead to challenges, especially when it comes to being able to scale components independently or flexibility around maintaining the system. That being said, these architectures can be very fast when designed correctly and responsive if it is designed for the needed load. They have far fewer trips across the network in most cases and are locally run with a lot of localized libraries versus remote calls. Ironically these libraries used to be called APIs, where many times remote services are referred to as APIs today. These architectures generally lack the ability to have remote objects that other applications can share and use; which is one of the main benefits of other architectures leveraging web services, remote method invocations and TCP/IP services. This is not a hard and fast rule, but it is generally true. When it comes to performance, the more calls going across network, the more latency the platform will endure for each request. Network latency depends a lot on region, network switches, balancers and network bandwidth at a given point in time. It can be as fast as 5ms and as long as 200ms across the same request with outliers as high as 2 seconds if there are a lot of packet retransmits in the communication stream. The same request will vary by the amount of variance across those devices and the network bandwidth usage. This latency varies based on the traffic and network bandwidth available as the packet's route across the different components. Other architectures, beyond monolithic, need to jump through a lot of hoops to achieve scale to compensate for these network times. Monolithic approaches to development may

translate to desktop or client/server application solutions. So, the database in *Figure 10.1* does not need to be remote to be considered monolithic.

Monolithic architectures achieve reuse by sharing software source code for libraries and packaged more native runtime components, libraries, etcetera. Though they may not be able to scale as well as the more flexible architectures that we will discuss next, it does not mean that the more flexible architectures will respond with less response time. The differences between capacity and performance are two different things, though when one is impacted, it may impact the other. With this architecture, software components are many times interconnected and interdependent, a feature that eliminates the need for inter-process communication.

When a system does not have to spend time communicating between many different services, encountering network latency and deviations in response times for web services under different load conditions and infrastructure, performance is enhanced because operations are quicker. This efficiency stems from the fact that data can be shared among different components of the system via shared memory and it is easier to manage the system's resources more optimally. When all of the software is running under the same memory, CPU, disk, etc., it is much easier to achieve less variance in responsiveness. Thanks to these characteristics, monolithic architectures are capable of delivering a great user experience. Monolithic architecture does not translate to outdated technology; it still has its place for applications where scale is known, and it can be created much faster with fewer resources.

Microservices architecture

To understand how many microservices are leveraged in a larger architecture, it's key to look at each layer independently.

Presentation layer: This is the user interface where interactions happen. From a website's frontend to mobile app interfaces, this layer impacts the user experience. You want fast performance here, caching, and streamlined loading of resources to minimize browser thread usage. You need robust observability here to ensure quick load times, responsive design, and seamless user engagement:

- **Microservices**: This layer is the core backbone that consists of loosely coupled small services. Each service has a specific function and interacts in many cases with other services across a network, usually in a RESTful fashion using the HTTP protocol. Observability in this layer involves monitoring service intercommunication, performance bottlenecks, and error rates. It helps in pinpointing issues precisely and maintaining efficient service delivery. We want to look for requests that are too fine grained with too many calls across the network or services with too many retries on failures.
- **Database**: Your data repositories need to be constantly monitored for performance and integrity. Observing database operations and

identifying slow queries, connection issues, and data consistency problems. It is not just about performance; it is also about maintaining data security. We want to know everything that is happening with the database. When things back up here due to slow or inefficient queries, they generally backup the entire layered architecture being used upstream. The best databases to use allow for stored procedures or have drivers that support multiple SQL/NoSQL statements.

- **IT User**: The IT users are the engineers, developers, and users who interact with these systems. Observing their interactions helps in streamlining processes, identifying pain points, and enabling them to respond swiftly to anomalies and incidents. To do this, analytics becomes very important, and observability around what the user is doing, where they are having trouble. Session replay is a good feature to have here for web applications.

Figure 10.2 represents these different layers. Understanding these layers helps in designing a comprehensive cloud observability strategy tailored to each component of your microservices architecture. Doing so ensures high availability, performance, and reliability across your entire cloud infrastructure.

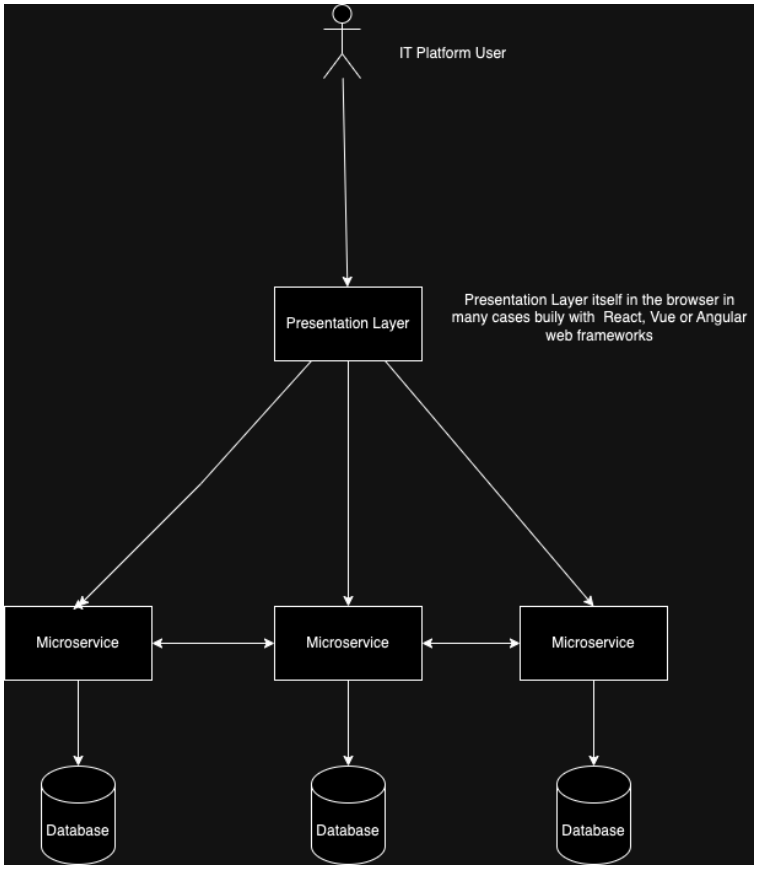


Figure 10.2

You have heard us talk a great deal about microservices with Kubernetes and event-driven architectures in this book. Microservices started becoming very popular when *Netflix* utilized them in their architecture. They were sold as something small and finite that could be released directly to production independent of other components in the application. This means you could leverage them for **continuous integration/ continuous delivery** (**CI/CD**) and frequent releases could be done reliably with less impact to the platform than with larger services or modules. Although this approach holds its own charm when it comes to developing and deploying large-scale applications through small, manageable services that can be elastically scaled in the cloud, there are some disadvantages to bear in mind.

The first thing to keep in mind is that managing a microservices architecture can be difficult. The process of managing multiple fine-grained services, each developed and deployed independently all managing their own database can be challenging to support. Synchronizing and correlating data between services is not straight forward and the integrity of data becomes a serious issue with microservice architectures. When combined with an event-driven architecture, this may also mean timing issues come into play around data and communications, further complicating fast resolution of issues.

Another drawback to consider is the need for extensive communication that needs to occur with these granular components. Your different services need to communicate consistently and reliably, which can lead to network latency or slow processing times if not well-managed. One of the advantages of platforms developed with micro-services is that multiple languages for those services can be used, but when different coding languages, frameworks, etcetera are used, it can lead to inconsistencies and confusion and ultimately slow down the development process. Though microservices can be elastic scaled services that can be tuned independently for capacity, this does not always translate to good performance, just flexibility around capacity in functionality.

The more requests that your platform makes across the network to services for information, the more interconnected communications you need to make across multiple fine-grained microservices, and the slower the response times. The more microservice dependencies you have, the more service contracts that have to be maintained across versions for testing. This can be challenging. When it comes to microservices, we see that most organizations do not take advantage of them properly, which negates much of their advantages.

Instead of having a development environment, quality assurance environment, and a **user acceptance testing** (**UAT**) environment to maintain identical to production, consider reducing the infrastructure you maintain using something called blue / green deployments. Develop using a fixed image locally or using a cloud version of a desktop and then go right to production for releases from the repository. If you do go down the microservices route, then best practices are generally to separate the data store, keep microservices code at a similar level of maturity, and separate builds for each microservice; they should have a single responsibility, be deployed into containers managed by Kubernetes for orchestration, be stateless ideally. UI services, referred to as micro frontends, should go

through an API gateway for interaction with backend services. Let us briefly go deeper into blue/green deployments and why managing service releases is so important.

A popular strategy adopted in a microservices architecture is where you can deploy your microservice directly to production. If you are not doing this or leveraging CI/CD, do not leverage microservices; you are not achieving the largest advantage of not maintaining so much infrastructure in multiple lower environments and costs associated with large releases. Shoot for continuous deployments in everything that you do if you decide to leverage this architecture and have teams take ownership at the service level.

Essentially, blue/green deployment involves running two identical services, referred to as blue and green, or two entire environments that switch in production. The blue is the live, production environment serving end users, while the green lies idle. There are many derivations of blue/green from isolated switchovers at the service level to entire infrastructures. When a new version of the microservice is ready, it is initially deployed to the non-live, green environment. Performance and functional tests are then run against quality gates, and any necessary checks are verified to ensure that the new software version functions as intended. Then, chaos engineering checks are run to attempt to break the new features.

After successful validation of the service a traffic router then switches all incoming end-user requests from the blue environment to the green environment. The previously active blue environment now becomes idle, ready for the next round of deployment. If you have very large releases, you can replicate the entire environment at an even courser level, one being blue and the other being green, and switch traffic at a course level from one environment to another entirely, but with small changes directly made to production with microservices, you can be granular at the service level. If you want fast delivery, you need to get away from the large amount of processes in lower environments and too many environments. The cost savings of taking this approach is something that is not understood, it can be a huge savings when you can eliminate the processes and environmental costs. If you want to run microservices, we would encourage you to make small changes and release often in the development process. Minimize dependencies and isolate services, including the data.

Using a blue/green deployment and release methodology gives you the ability to quickly rollback changes by simply switching back to the blue environment if any issues are detected in the green environment, minimizing service downtime. Thus, the blue/green deployment strategy provides a safer and controlled way of releasing new software versions, though it requires two production-ready environments to work seamlessly or, at a minimum, a pair of microservices being worked with each operating with the other dependencies in the real production environment with a test data until the switch over.

One good approach here also is to test the new microservice being developed in the green environment and then start routing 1% of the production traffic thereafter the performance and functional tests pass if you have this option. Then, if all is good after a period of time, switch that service with the blue service and make it the new version.

Netflix popularized the microservices architecture significantly and drove a lot of companies in this direction, though some services look closer to an SOA paradigm when reviewed in detail. Development teams store data in a NoSQL database for most services, with the exception of billing information, which goes into a relational database. On top of the microservices layer in their architecture, they use Memcached and pull from the cache vs. going to the microservice to speed up requests across microservices and the backend. So anytime one service calls another service, the cache is always hit first and the response returned for the request. The same goes for database query requests; data is stored in the cache for frequently accessed data. They leverage Kafka for queue event handling and process 2 billion transactions each day.

Netflix will deploy software hundreds of times a day, sometimes thousands. They leverage no quality engineering, site reliability engineering resources or have any release engineers. How do they do it? *Netflix* uses **platform as a service** (**PaaS**) technology to allow them to deploy parts of the infrastructure for a service whenever they want no matter how many times, they require it and leverage blue/green deployments. This is not new technology, technology to create an image with the infrastructure and software created in one go has been around for many years now.

Netflix has a separate development team working on each service, which means each team works autonomously and decides for themselves how often they want to deploy their code. They, on the fly, create a new infrastructure representing the service they are going to deploy, representing the green side of the blue/green deployment. They spin up the new environment for the service, and performance, and quality test it against quality gates, automatically run chaos engineering against it on the green side of the equation, and then switch traffic between blue and green for their service and make the service live.

To begin the process, first, before any deployment happens, the developer will first push the code into a source code repository like GitHub. The code push triggers Jenkins to initiate a software/infrastructure build, producing an application package and creating a fresh *Amazon Machine Image* (Virtual machine) containing a Linux distribution, Kubernetes, the Docker packaged software being released, the JVM, etcetera. Testing occurs against the green deployment once the machine is started with the service running (performance and functional), then traffic is routed from blue to green. This whole process is automated and very fast.

Netflix has even built out its own **content delivery network** (**CDN**) infrastructure using Golang leveraging the languages concurrency features to achieve great performance. Essentially using isolation techniques with microservices and letting teams run isolated and autonomous is what allows them to execute small features so many times a day. They have an SRE team that has built out this PAAS capability of building the infrastructure needed at a service level and all teams leverage it and common caching and event infrastructure in place.

When it comes to microservices and implementing them well, combining different patterns can provide comprehensive solutions, enhancing

overall efficiency of your microservices. Among many powerful design patterns, some of the most effective include:

- **API gateway pattern**: Single point of entry that manages requests/responses for a set of microservices.
- **Circuit breaker pattern**: Prevent system failure in case of network latency or non-responsive microservices. If one service fails, the circuit breaker stops all requests to that service.
- **Sidecar pattern**: Enhance or extend s service without needing to modify the core service. A sidecar will run next to each micro-service, providing added functionality.

High-performance cloud architectures leveraging microservices when integrated with Docker and Kubernetes provides several benefits.

- **Independent deployment**: Microservices can be deployed independently like *Netflix* does. The development process speeds up and boosts productivity not to mention granular scalability.
- **Data security**: Since each microservice has its own separate database, it adds an extra layer of data protection.
- **Flexible service creation**: Different teams can work simultaneously on various microservices using different technologies, increasing the speed and flexibility of the development process.

Though we have discussed some of this previously, we want to reiterate that when microservices are packaged by Docker and orchestrated under Kubernetes, together, these technologies play a transformative role in achieving high-performance cloud architectures. Docker ensures each micro-service runs in its own container with a private runtime environment. This reduces the chances of conflicts between different services, thus reinforcing application resiliency. Kubernetes takes care of the orchestration of these containers, ensuring they work effectively together. It takes care of service discovery and load balancing and automatically handles the service elasticity, scaling pods up or down as required. These technologies create an ecosystem for your microservices, where services have the freedom to perform independently.

Docker

Docker is a platform that establishes containerization to package an application and its dependencies into a standardized unit for software development. The beauty of Docker lies in its simplicity. When software runs in containers, you do not have to worry about dependencies and environmental inconsistencies. Applications packaged in Docker will run the same in every environment. Docker brings many benefits to the world of microservices and cloud architecture. Its features fit nicely into the microservices architecture for a few reasons:

- Docker ensures application isolation where each microservice runs in its own container with its dependencies.
- Docker containers are scalable, enabling you to increase or decrease the number of packaged containers at runtime.
- The platform helps facilitate continuous integration and deployment.
- Docker accelerates development and deployment times. It also developers to replicate the production environment on their local machines.

When Docker with Kubernetes are used together, you have a recipe for a high-performance cloud architecture that can achieve a great deal of scale. Kubernetes can manage a cluster of machines and schedule containers to run on them based on their resource requirements and other constraints, providing pod fail over capabilities when needed, providing web service resiliency.

Service oriented architecture

When it comes to a web services discussion, another type of architecture is called a **service oriented architecture** (**SOA**). It is not a new player in the field and can almost be considered a predecessor to microservices. But essentially these are coarser grained services designed many times based on business domains or functionalities which can be used across multiple applications. Some use the REST and others use the SOAP protocol. For example, one service may be called a **payments service** or another a rewards service, that handle all of the business functionality and transactions for these types of transactions. We remember that AT&T used to leverage an SOA architecture with the concept of **business application services** (**BAS**) which were referred to as software applications of all times and **business integration services** (**BIS**) that were SOA services that performed various functions in the telecom domain from product management to resource management around procurement of services.

Enterpriser service bus

Generally, services like this are orchestrated by using an **enterpriser service bus** (**ESB**) or a pub/sub messaging bus and different business domains across the organization can subscribe to the bus for integration. Applications, multiple different types from various sales channels referred to as BAS services, would call the BIS services that communicate with one another through the ESB.

Using an ESB was an important step for high-performance architectures. Initially, when web services first started, they called each other directly, which led to all types of service communication issues and failure problems. Imagine each part of an organization developing their own services and calling everyone else's and trying to manage performance, capacity and orchestration. Before the ESB came along there was the concept of a hub and spoke architecture. It was a central hub for all of the services to go through

and was always reflected as a big bicycle wheel in diagrams. The problem was that if the hub went down, everything went down, but it was a similar concept to the ESB and is really a pattern and concept if you think about it, though many implemented it as a standalone IBM MQSeries implementation when it first came out. It managed communications from a central point. ESBs went further with more clustering and orchestration capabilities to facilitate the Hub and Spoke pattern without a single point of failure. They are in high use today across almost every large organization.

Service orchestration

SOA services can also become complex in large architectures due to the need for service orchestration and governance, but generally, in a typical application where you have 5 SOA services that are coarse-grained, you may have 20 microservices which means the complexity is less compared to micro-services. SOA services, though larger with more dependencies and a centralized database in most cases, are generally faster than microservices due to less orchestration and less calls across the network in general to achieve the same functionality. They do not maintain isolation and management of the data layer. They will query or update a relational or NoSQL database directly that has relationships defined that are leveraged across all of the SOA services. Thus, no data integrity issues occur like microservices have and when you have data in a central database vs fragmented and isolated through service layers, it means large batch processes can be run without making requests across the network, which is one of the most inefficient things that we have witnessed in recent years with respect to batch process. That being said, it is almost a necessity with microservices architectures where services act as an abstraction layer over their database, and correlation between tables and services needs to occur.

Comparing SOA to microservices

Comparing SOA to microservices, a website may reach out to 5 SOA services for the entire application hitting a relational database vs. interacting with 40, each maintaining their own database/views/tables not connected to other service data dependencies, which translates to much more network latency and service to service interaction and finer grained calls to the database. Though SOA services cannot be scaled as granularly as microservices, they can be coarsely scaled in the same fashion. SOA services are popular in high performance layered architectures. If you are transforming away from a monolithic architecture to something that scales better, an SOA architecture can be an easier transition, though not as granular as microservices for scaling, this approach can elastically scale at a coarser level very well.

If you are not doing CI/CD with microservices and taking advantage of doing them the right way, eliminating infrastructure, etcetera, autonomous releases with owned services by single teams, then using SOA is a great way to speed up your application and get better responsiveness. Just like monolithic architectures call libraries locally with less calls across

the amount of service dependency calls across the network. If you are looking to leverage continuous delivery where code can be delivered at any time during the day or multiple times a day, consider the smaller units of work that are in micro-services architectures.

Another type of architecture prevalent in the cloud is to use a serverless architecture or managed services. If you do not have strong talent or you want to speed up product creation, this is not a bad approach though it does lock you into the vendor. In a serverless architecture, developers are free from the task of dealing with their own servers. Instead, you use cloud services to execute server-side logic and manage databases. Since you rely on third-party cloud services, network issues and function execution time may become challenging.

Once you choose a provider for your serverless architecture implementation, it can be quite challenging to migrate your services to a different provider due to non-standardized technology across vendors if you would need to do that. Something else to consider is debugging and monitoring**.** These can be more complicated with serverless architecture. These architectures inherently lack a standard, continuously-running environment where traditional fine grained controlled monitoring solutions can be leveraged. Additionally, you may face issues with latency. With serverless, servers need to start up and load your application before they can handle requests.

This cold-start for managed services and serverless architectures can cause small delays, especially if your infrastructure is not in constant use. There is also a potential for unexpected costs as you sometimes pay more for the execution of something being managed by a third party. A sudden increase in usage, can greatly increase costs. Anytime you leverage service managed by a cloud vendor called managed services or leverage serverless architectures where the runtime can be ambiguous with vendor lock-in, consider avoiding it if you need flexibility.

Managed services are where the cloud vendor hides the complexity of using solutions like Kubernetes, and provides their own, easier to use services that do not have any portability across cloud providers. The benefits to Managed Services and serverless architectures are that they help achieve fast development but remember that they are black box when it comes to performance optimization. You do not have as many knobs that you can turn to make it faster or reduce outlier latencies being seen beyond the vendor login issues.

Event driven architecture

We have talked a lot about an **event driven architecture** (**EDA**) in this book already, so we will go light on this topic in this chapter. EDA puts events, state changes at the center of the design, these events trigger the functions of components within the system. This is one of the most flexible architectures that can be created, it can also be one of the most chatty architectures out there that generates an enormous number of requests across the network with timing issues and collisions when not properly implemented. Combine this with data consistency issues with microservices can be challenging and be difficult to support. We

do not want to give the impression that EDA solutions do not scale, because many times requests across the wire are asynchronous events that when properly timed can result in a good outcome. But always consider the number of connections that need to be active, requests across the network, and the potential for additional latency being encountered; along with support complexity and costs. Many organizations use service buses with SOA services to integrate team software for large platforms. EDA is an evolution many times associated with microservice event driven design patterns though it does not have to be. When it comes to architectures and services, consider supporting multiple protocols and give yourself a lot of flexibility without future changes. APIs that can be called in a synchronous or asynchronous fashion for example. It is important to standardize architectural patterns but support multiple protocols, payloads, events, etc. across your APIs to allow for unlimited flexibility. Also build or support your own transaction management including rollbacks across services in anything that is in the architecture. Just because an architecture uses micro-services with each service having its own database does not mean that a facade layer or delegate on top of the service layer cannot manage the transaction and roll it back. It is important to build in rollback capabilities for all APIs and services out of the gate and not make it a secondary consideration.

Data centered architecture

Another architecture type is called a **data centered architecture** (**DCA**) where data frequently accessed by other components is centralized. This makes data storage and access a priority. This can breed a very fast architecture and is a great approach for performance when data consistency is required.

Client server architecture

Client/Server architecture is one of the early architecture styles that expanded upon the monolithic architecture to break out the front end of applications to the backend. So, multiple frontend applications could access services on the backend for business logic. This division of labor allows for a distributed system, handling tasks efficiently and offering high capacity for expansion or upgrading. It is a popular architecture used in network computing where software applications are split between server tasks and client tasks. It has been an important precursor to Layered architectures.

Layered n-tier architecture

Layered n-tier architecture is one of the most popular architecture styles out there. This architecture allows for the separation of concerns across layers, making it easier to test and modify one layer without impacting the other, and is one of the easiest architectures to maintain.

In this architecture, for good performance, caching occurs at each layer and this is one of the

when leveraging coarse services. Most larger companies that need many applications interacting will have applications running distributed architectures that are layered and leveraging SOA representing a domain or function for service interaction. They leverage a clustered hub for async or sync events to orchestrate component interactions and a relational or NoSQL database on the backend. These architectures many times use a common database and perform very well with separation of batch standalone programs and real time services. They are easy to maintain and augment.

The finer grained you go with an architecture, the more flexibility you have. The coarser grained you go with an architecture, the less flexibility but the greater the simplicity when it comes to resolving issues and developing new capabilities. The architecture to choose depends on your needs for a project. Do not assume that all software should follow the same architecture. It is always a balance of flexibility with support, capacity, and performance.

Figure 10.3 depicts the different layers of an n-tier architecture design. Like all architectures presented in this book, observability at each architecture layer is crucial for performance optimization.

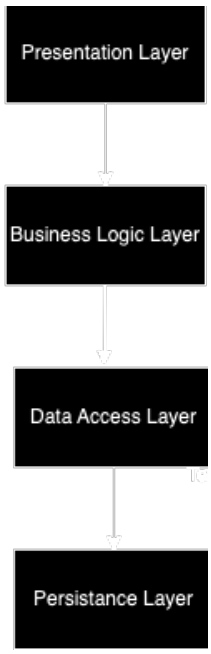


***Figure 10.3**: Layered N-Tier architecture*

Another less popular architecture style is called a **space-based architecture**. This architecture is all about avoiding data contention issues and bottlenecks by smartly leveraging replication and partitioning mechanisms to achieve operations without a collision. This architecture is considered a very high-performance architecture than can be

The last well-known and frequently used architecture type and design pattern we want to briefly mention is called the **microkernel architecture,** which essentially allows developers or system engineers to update or add components without impacting other functionalities through isolated units or plugins. It is very popular for designing operating systems and servers. To achieve great performance, especially with Linux platforms, we would recommend you reference the materials by *Brendan Gregg* here:

https://www.brendangregg.com/linuxperf.html

Sometimes, when considering what architecture to use, it is important to see what others are leveraging. When we talk about case studies around performance, there are a lot of good examples of where good architecture helped achieve great performance and scale. One of the best places on the web to look at these is one of our favorite sites, highscalability.com. Here is where you can learn and see how other companies are achieving great scale:

http://highscalability.com/blog/category/example

When it comes to the Cloud and transitioning to great scale, a few organizations come to mind. One is Panera, who moved from on-premise Linux servers to the *Google Cloud* and used managed services, Java, and NodeJS to achieve great scale in the cloud with terrific performance. Equifax and MasterCard are other organizations that have made significant strides in cloud transformation. Another example is Airbnb. They had a problem with being able to handle large data surges during peak seasons and were able to dramatically improve their customer experience and availability by leveraging the elasticity of the cloud for scale and with *Amazon* hosting their infrastructure. *GE Oil and Gas* division was able to save IT costs by 52% by moving to the cloud and this is the same story with many companies. Instead of purchasing servers, managing updates, etc., they leverage the cloud.

Netflix was one of the companies that all of the rest have tried to follow with their forward-thinking approaches to architecture. They have more than two hundred million customers. When they took the time to shift to cloud-based platforms and CI/CD with microservices, they were able to independently scale those services and be able to release new features extremely fast, supporting a huge amount of scale. Pinterest has close to five hundred million users. They leverage *Google Cloud* and manage large volumes of traffic, and have more reliability that ever before.

One thing to note is that the architecture depends on what is important, such as data consistency, flexibility, fast releases, or larger, less frequent releases. When we see organizations that process payments try to replicate architectures of social networking sites, many times, these are not the right course of action. As with all design decisions, finding a good software architecture for your project requires considering many aspects of what an organization really needs now and what it will need in the future. It is not a one-size-fits-all scenario. The choice can significantly influence the efficiency, scalability, and overall success of your software. Here is a quick guide to help you steer your way through the selection process.

Start by looking at the requirements for a project and do not take a one size fits all for all applications you are building. Think about the goals of the effort and the capabilities that need to be delivered. For example, a shopping website that may have features and content changing daily with promotions impacting traffic surges and patterns that can be unpredictable may see some benefits from using a microservices architecture with continuous delivery so that individual services can be scaled at a very granular level by actions occurring on the platform.

Perhaps a basic web application that has predictable patterns where, once built, features will not change frequently and ongoing can benefit more from a monolithic architecture. The more complexity of the architecture, the more difficult it will be to support and maintain. Resolution times for issues go up, so this is an important consideration.

Think about the technology stack at your organization and the team's expertise and knowledge. Learning a new architecture can kill productivity. If most of your resources are experts in layered n-tier architectures, consider if you want to disrupt productivity to build something the organization is unfamiliar with. This is always a risk, whether or not that risk is worth it depends on how important change is and if the additional costs can be incurred.

Remember also that in any architecture that is chosen that interoperability between other platforms is important. Another important consideration is growth and feature frequency ongoing after the initial product or platform is delivered. If you expect data volume and users to expand significantly in a small amount of time and you have specific functions that users leverage that have a lot of traffic, then having granular capacity expansion through serverless, and microservices may make sense. But also consider operational and maintenance costs.

Do not over engineer your IT platform! Consider how much the ongoing maintenance costs will be due to the complexity, consider time and effort around resource availability. The architecture chosen depends on the project's specific needs. Do not try to replicate the same architecture for every application unless there is a significant benefit to doing so.

Separating batch and real-time

Another basic consideration when it comes to developing high-performance architectures is to separate your batch processes from real-time to eliminate contention. With the advent of microservices, some architectures will use the same infrastructure and same services that are used for real time traffic. Since with microservice architectures many times each service has their own database to manage, it means the services act as an abstraction layer on top of each data source and correlate data. This can become problematic because processing records or batch manipulation of records and data requires service calls across the network at a granular level.

For example, if you are processing records such as with a profile, you may find that you need to reach out to multiple services for address

data, etcetera This means to process a single profile record, you need to make an enormous amount of calls if you were needing to process 100,000 records. For large global organizations that have a large number of profiles and customers, like social networks, the phone companies, etc., this may mean 500 million customers. This contention generally puts a strain on both the real time traffic and your customers in addition to impacting the time it takes the batch process to complete. When processing 500 million records, you need to be able to do this not using a microservices architecture but instead database technology and a common database when it comes to large batch volume like this. These considerations influence the architecture selection process.

One of the advantages of a layered, monolithic, client server, SOA, etcetera type of architecture is that you have a centralized database with all of the proper table relationships, constraints, indexes, etcetera and instead of needing services to hit the data with an abstraction layer, you can write batch programs that process database records extremely fast using fast languages like Golang, RUST or C language. Instead of impacting nodes and clusters in your infrastructure and generating large amounts of network traffic or experiencing network latency with every single request, you can have a vertically scaled VM machine that handles the fast processing of database loads, extracts, or large amounts of work related to data manipulation. You directly have the batch program talk to the database. Of course, going horizontal or having multiple batch servers is an option, but the point is to not go across the network for records you are processing in a granular fashion using a language like Java, Python, C#, or other. Instead, write the batch program using a language that is native that does not require a JVM or abstracted runtime layer with garbage collection.

Pervasive design patterns

Regardless of the architecture chosen, many times, a component of that architecture is a web user interface that your customer leverages to interact with your IT platform. We have talked some about caching, browser caching, CDN, etcetera. However, the way to achieve the absolute best scale is to leverage pervasive computing, leveraging HTML5 and web service workers, and make the application run almost standalone, only going across the network when absolutely required. In other words, the application should run in the browser, just like a desktop application. HTML5 became popular initially because it allowed you to play video and audio files without third-party plugins, which significantly improved the performance of applications and brought consistency around performance expectations. It also allowed new elements (header, footer, section, article) to help structure content for search engines. But it is so powerful. Here are some of the capabilities:

HTML 5 and local storage sync patterns

Advantages of using HTML5 in an IT platform:

- HTML5 offers better connectivity with communication APIs

- It enables access to the device's hardware, camera, geolocation, accelerometer, etcetera
- It can handle graphics easily
- HTML5 supports multimedia elements like audio and video natively
- It has features for fast speed, system integration using web workers, localized databases, and application caching
- Desktop application capabilities as far as UI interactions
- HTML5 offers web storage which persists across sessions after the browser is closed. No more cookie management

Let us say that you are developing an application that hires people and you need to enter data and generate 10 PDF forms as part of the hiring process. You decide that you are going with a pervasive approach for your UI journeys. This means that you would want the platform to run standalone as much as possible on the user's desktop and only periodically connect to a central site when you are online. We cannot put enough emphasis on how fast this is for a customer to use your site and how much less scale you need due to fewer traffic interactions across the network with coarser-grained requests. So, you use the persistence data storage capabilities of HTML5, which all major browsers support these days.

In HTML5, you have the choices of Session Storage, Local Storage, IndexedDB and the File API, which are great options for pervasive computing both offline and online. IndexedDB is a capable transactional database that allows for the persistence of large amounts of information. It ensures data integrity even when multiple threads are accessing the database. Like Local Storage, which is a lightweight persistence mechanism that can persist data when the browser is shut down or across sessions, IndexedDB can do this as well, except with even larger amounts of data and safety.

Localized data security

When encryption is required, you leverage CryptoJS or Web Crytography API for web browsers. So much power and security exist these days. In the example we have set forth, you enter the data on forms that are cached locally in the browser that run independently of an online service, with the exception of syncing versions for content changing. Then once entered, you generate the forms in PDF format and store them as blobs in IndexedDB. You leverage a library like **`jsPDF`** to do this. Then, once you have the 10 forms generated, you submit them to the central server when online. Let us say perhaps you need data online to generate the 10 forms. In that case, you enter the data needed for the 10 forms and then in a single coarse-grained request, retrieve the data needed for those 10 forms online.

Syncing data

This is how you create extremely fast applications. Perhaps another obstacle is that the screen where you enter the data needs an update. What you do is when the application starts or when it becomes online, you have a version stamp for the application released and a version stamp for the change in the cloud and then sync up the changes to the pervasive application during the online window.

Perhaps the user already entered data into the old form before the sync happened; in this situation, only prompt in the application for what is missing and ignore the rest. The major point to remember is that pervasive computing, allowing interactions locally like a desktop application in the browser without connectivity, is how you create extremely fast applications and happy customers. It also, most importantly, distributes compute power to the user's machine and eliminates the run time costs you would encounter processing the data in the cloud. Remember that today, with both mobile and desktop, all major browsers have been supporting HTML5 for a long time now. They all have the needed capability to persist data and operate in a secure and pervasive manner where you operate like a desktop app and only periodically make requests across the wire to fill in gaps of information.

Offloading UI tasks and past UI challenges

You have heard us mention web service workers in previous chapters to improve web performance. These workers function as a HTTPS proxy between an application, the browser and the network when it is available. These will offload tasks off the man thread being used for processing in your application, making your application perform much better. It is like multithreading your web client. HTML5 is your friend when it comes to creating high performance architectures needing a web frontend.

The challenge initially with desktop applications was that support was difficult. Everyone had all sorts of different applications on their local machine that would interact with what you were installing, making support a nightmare. When this occurred, everyone wanted everything to be online, and you could not trust browser consistency, so all requests went back to the server for content, and the complexity of the user interfaces was limited. Programming languages like Java tried to improve capability with Applets, and *Microsoft* tried to push ActiveX controls that were similar to get a desktop feel, but acceptance was limited.

Lightweight simplicity

The absolute fastest websites we can remember were just basic Java Servlets that used a **`ResourceBundle`** class to storage HTML in its entirety, yes, the full HTML in the bundle (a key value pair with the page name, and html content with **`\n`** characters at the end of each line). Then, when the Servlet started up, in the Init method, a thread was spun

to cache all of the HTML for every page. In other words, the entire site would run from memory. Dynamic content was put into the html using tags with regular expressions. This old web development approach was extremely fast and lightweight. When a request happened, the website came back to the browser almost instantly. Our thoughts are that even *Google* could improve its speed by taking such an approach that is not well known at all or considered these days.

Today, we have the capability of having the beautiful desktop experience completely in the browser running locally without the worry of third-party installations impacting the application itself, forcing a support nightmare. It does mean good testing across devices and browsers and making sure that minimal resources are needed for a good experience.

If you cannot run pervasively for some reason, then cache at every simple layer in the infrastructure to minimize requests going to the database and select a database solution that allows for stored procedures or functions that allow for coarse grained requests where one request can execute multiple SQL statements. Many drivers also support a similar capability around executing in batch, but the database needs to support it. Otherwise, you will be sending too many requests across the wire, which will prevent the platform from scaling.

Cloud database selection

As you consider high-performance cloud architectures, one important decision that you will need to make involves your selection of a database. The correct database can significantly bolster the efficiency, speed, and adaptability of your cloud applications, while an inappropriate choice may result in poor performance, increased costs, and scalability problems. Relational databases, such as MySQL, Oracle, and PostgreSQL, have long been the standard for many applications because they support stored procedures. With a structured schema and robust support for transactions, these databases can guarantee data integrity and consistency.

Despite their advantages, the intricacy and rigidity of their schema can, at times, inhibit flexibility and scalability, particularly with high-velocity data. We have talked about this in other chapters. Solutions like MongoDB provide some great high-performance solutions. This is what organizations like Panera leverage to achieve scale for their e-commerce platform. For applications that demand strong data consistency, such as financial services or inventory management, traditional relational databases do well. However, if you can sacrifice a degree of immediate consistency for high accessibility, then NoSQL databases with their eventual consistency model are likely a good solution. We can tell you that many large companies operate with PostgreSQL, Oracle, or MongoDB. Many times, the cloud database solutions that are readily available for a cloud, though evolving, require too many requests across the wire, which helps the cloud providers' revenue but not the organization that is creating the product. They also are another mechanism to lock you into that vendor. Make sure that you consider the expense as well, the total cost of ownership, and infrastructure/database management.

Improving API performance

Improving API performance in architectures is one of the most important aspects of scaling a platform. The first thing to do is cache responses. Caching can dramatically decrease the load on servers and speed up responses from an API. Also, leveraging data sources that can store frequently retrieved data in memory will speed up requests. Another item required for good performance for an application is the concept of pagination. Pagination means not retrieving all information from the database or other services all at once. Instead, request the information in management chunks to allow the calling application to receive fast responses. The latency without paging is not just about the request latency across the network and I/O streaming back to the requester, it is also many times about query times in the database and the slow parsing from converting from one datatype to another. The larger the dataset, the slower that parsing occurs, the slower the mapping occurs when moving data into objects, and the slower the response back to the UI.

Another tip for fast API performance is to use async logging and **first-in first-out** (**FIFO**) buffers to reduce latency around writing to disk. Most logging frameworks support async threading, but many people do not know that they also support buffering of requests so that the async thread can return immediately by writing the log to memory with the payload in the buffer, writing to disk at a slower rate.

One of the things we see that is most often forgotten for APIs is the importance of compression of data going across the network. The smaller the payload, the faster it will transfer across the network and the less bandwidth consumption. But it is important to understand that you want to compress only requests larger than 5 Kbytes in size, or the CPU consumption and timing to compress/decompress will start impacting the benefits around responsiveness.

One of the quickest ways to slow down a web service is to force each request to spin up a connection and negotiate around SSL for security, DNS looks, etcetera. Spinning up new connection threads can be slow; this not only refers to requests across a network but also to connections to the database. When it comes to communicating with databases and services one of the best approaches is to have connections spun up and already connected to the database or service and using a pool of connections. When a new request comes in, you direct it to a connection from the pool, and it grabs an already spun-up connection that sends the request. Remember that when it comes to connection pools for third parties, you see a min pool and max pool setting. For example, 10 is the min and 100 is the max, that every time that pool grows it encounters latency. For the most consistent performance and fastest response, always set the min and max to the same value.

Lastly, load-balancing algorithms and rule filters can have a very large impact on API performance. It is important to select the correct algorithm for balancing. When all servers are equal in compute and memory, using the same CPU family, bus frequency, etcetera then round robin typically is the way to go. If not using the same exact servers and CPU family, bus frequency, etcetera, then the least time or least connections is a good approach

for the best responsiveness. From a performance perspective, the goal is to minimize filter rules, redirects that impact latency around requests and eat CPU for the balancer and route the request to where it will be processed the quickest with connections up and running (KeepAlive). Here is the full list of balancing strategies. We encourage you to become experts in these:

- **Round robin**: even request distribution
- **Sticky round robin**: keep request sticky by requester back to the same server
- **Weighted round robin**: favor certain servers for requests over others
- **IP/URL hash**: Route requests based on an IP or a URL
- **Lease connections**: route requests to servers that have the least connections
- **Least time**: route to servers than can process the request fastest based on load

We talked some about what companies have done, such as *Panera*, *MasterCard*, *Pinterest*, *Airbnb*, *Netflix*, *GE*, etcetera. But let us give some examples of thought processes when it comes to architecture.

Consider a case study where an organization needs some database optimization. They face challenges with older monolithic architecture and are experiencing some issues scaling their data and managing it efficiently as the platform grows with more users. Let us say that the data for this application requires continual updates and expansion due to new customers coming onto the platform for an organization that is growing fast. With a traditional monolithic architecture, changes for growth require a lot of downtime and the risk is data inconsistency issues as data is expected if you scale quickly, a lot needs to be considered around risk. To overcome these issues, the organization decided to move to the cloud and adopt a microservices architecture with a NoSQL database, Docker, and Kubernetes for elastic scaling capabilities and resiliency.

To do this, it was important first to transition their monolithic structure into containerized microservices using Docker running in Kubernetes. Let us say they need to be extremely fast, and data consistency is not as important as speed and flexibility in data structure. Thus, they choose a MongoDB NoSQL database to use. With Kubernetes managing the services, they were able to scale up during peak times and scale down during off-peak hours, increasing cost-effectiveness and providing seamless user experiences. MongoDB gave them great flexibility and scaling as well operating much of the data in memory and scaling horizontally in realtime. The result is that now this organization is able to handle peak user traffic without any data issues with minimal downtime improving customer satisfaction. Some important points:

- The move towards microservices resulted in better management and easier updates of specific services with minimal effect on the whole system.
- The use of Kubernetes for orchestration seamlessly bridged Docker containing microservices and the fast and flexible NoSQL database, allowing efficient scaling and optimizing resource use.

- Choosing a NoSQL database like MongoDB offered superior scalability and flexibility, making it a preferable fit for the distributed nature of microservices.
- Every component in a high-performance cloud architecture, from the database to the deployment environment, should be chosen strategically based on specific needs and goals to ensure optimal performance.
- If you choose a complex architecture such as microservices and plan to leverage the flexibility around granular scaling, make sure you invest in a lot of training in the organization. This is not optional. Training and educating your team on innovative technologies like Docker, Kubernetes, and microservices are key to being able to support the architecture and troubleshoot issues.

Conclusion

In this chapter, we discussed the different architecture types and the advantages and disadvantages of each type. We talked a great deal about case studies and what other companies are doing as far as architecture and achieving scale in the cloud. We also talked more about how to speed up applications from both the front end and API layers in the architectures presented. Beyond this, the importance of separating real time and batch infrastructures was discussed.

In the next chapter, we will go deeper into logging and error handling to ensure that the software remains fast during unforeseen events and that performance issues can be resolved quickly. There is nothing more important once a good architecture is established than having good error handling and logging to assure that the platform can overcome issues fast.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 11
Error Handling and Logging

Introduction

This chapter of the book will talk about logging and error handling. What does log and error handling information have to do with performance? It has a lot to do with contributing to performance latency and, on the flip side, reducing **mean time to repair** (**MTTR**) for performance issues on IT platforms. Verbose logs and generation of stack snapshots are some of the most expensive things from a performance perspective that can occur. An exception thrown with a stack trace can be the equivalent, on average, of 100 lines of code execution latency; it has that much of an impact.

Excessive logging and I/O related to presenting log information in consoles and logs can be detrimental to performance running a large number of threads in the JVM, tying up resources and slowly backing up the synchronous threads that are passing off the message to the asynchronous threads in the logging framework used. If those messages are being written directly to disk and the asynchronous logging threads backup even further, the heap memory is being excessively used and the garbage collector is continually context switching between the threads in order to clean up no longer referenced logging class instances. This is very much a misunderstood topic and one that we need to handle when implementing an IT platform.

Structure

This chapter covers the following topics:

- Introduction to logging
- Error handling
- Log standardization and documentation

Objectives

In the last chapter, we talked a great deal about architectural considerations and performance. We focused on using the right architecture for the right job and the advantages of each architecture. In this chapter we are going to discuss logging and exception handling in depth and talk about performance impacts.

Now that we have come so far into discussing good design patterns for performance around architecture, it is time to talk more about lower-level implementation mechanisms supporting the product that has been developed. We will discuss different logging levels, exception handling, and how to have applications log with great performance. Readers must understand Java development to understand this chapter as a precursor to reading this chapter on logging and error handling.

Introduction to logging

Cloud computing has changed greatly how we manage and operate IT platforms. However, we still see a lot of fragmentation around logging and tracing due to legacy tools being leveraged in the cloud or multiple tools being leveraged for the same observability. This not only generates confusion; it generates many times different results, bringing doubt into what is being seen.

When it comes to logging or exception handling, the first thing to remember is that in any organization, we need a single pane of glass as a starting point. One of the most important aspects of maintaining the health and performance of IT platforms involves proper logging and error handling and being able to support the product quickly, if we need to go into 3 or 4 observability tools to understand what is going on, that becomes problematic with respect to restoring a platforms proper operating state.

In the context of observability around software and infrastructure for IT platforms, a single pane of glass refers to a single unified view where all necessary information is presented in a clear, consistent, and informational manner. It helps us keep a pulse on the system's health, and manage resources effectively. The advantages of a single pane of glass approach in query log information and performance over time include:

- Information in one placemaking it easier to monitor and manage.
- It facilitates quick analysis, allowing us to respond quicker.
-

To implement a single pane of glass:

- Create the dashboard in a platform that allows us to retrieve the data from different sources (Grafana, Splunk, Looker, etcetera.) that has some flexibility in dashboard design as far as how to present information.
- Identify all data sources to retrieve the information needed that should be integrated into our unified dashboard view.
- Configure the platform to gather data from the identified sources.
- Design the dashboard to be high level with the ability to drill down easily into the issue even if it means drilling down into another tool handing off the observability from the dashboard. Do not create a dashboard that has high level information that does not handoff to a mechanism for a deeper drill down. The goal is to address issues fast, identifying the problem fast is the first step.
- Test how long it takes to find the issues over time and if the dashboard needs to be further refined.

Error handling

Improper error handling can lead to unnecessary complications, loss of data integrity, and severe consequences for system performance. It can also lead to unnecessary time being wasted trying to debug an issue without clear and concise information being represented around observability. Logging is essential for debugging issues and maintaining a healthy system. It offers a granular view of what is happening in our application at any given point in time that represents both the frontend and the backend in most cases. However, when excessive logging or deficient error handling occurs, it impacts performance and run-time costs.

Efficient error handling can be a strong differentiator between a well-performing application and a poor user experience. Error handling and proper level messaging back to the user will help determine a great user experience or a troublesome one that causes bounce rates to increase and closes the opportunities with new revenue streams as customers stay away.

Actionable logging

We know that we need to perform logging, but what information do we need to log? As a practice, we only want to log items that are actionable. If we are throwing exceptions and logging items that are not actionable, we need to consider the value proposition around what we are doing. Logging is not about tracing or debugging in production; it is about identifying the root cause and where it came from just by the exception itself being thrown. It is fine to have different logging levels if we want to incorporate tracing capability into the application, but it should not be enabled normally when the platform is running. To enable a debug level trace, it should mean just flipping a value in a configuration file or database. Remember that if we use the database, we will likely enable debugging across all

Exception handling with Java

Java is an object-oriented computer language that has terrific exception handling capabilities. The language leverages try, catch and finally code blocks to enable developers to capture exceptions. We can even have multiple catch errors defined in a single block statement in newer versions of Java. When an error occurs, exceptions are thrown that indicate the nature of the problem in the application. Logging should have standards defined in an organization, and the log text should represent the error that occurred, why it occurred, what source code, class, and method ID the exception happens in, and the data passed into that method.

Making the IT platform supportable

Why do we need logging standards? So that alerts can be triggered based on the logging that is occurring and not on things like HTTP response codes from backend services. Let us be honest: those return codes do not go deep enough into the issue or provide any information about what is occurring to drill into the root cause immediately.

In many instances, due to industry practices, software and hardware errors like HTTP 500, or HTTP 404 tend to meld between both infrastructure and services. This is one of the most time-consuming issues around support that we have seen in recent years and an awful pattern that has had a bandwagon effect, with everyone jumping on board due to blogs on the internet. When a 500 message comes back from a service, is it because the reverse proxy or balancer could not reach the service, or is it a software error itself with an unexpected result? When we query a database and get back a 404, is that because the URL was invalid that we sent to a balancer, reverse proxy, or is it because the software decided to use 404 when no records were returned?

Though using HTTP codes for software for some reason has become mainstream in recent years due to simplicity as a pattern, we would encourage anyone to think outside the box and do what is best to support the IT platform. For web services, balancers, reverse proxies, API gateway appliances that we do not have control over in our infrastructure as far as the underlying code, let these return these HTTP codes. When we see these, we know that we definitively have an infrastructure configuration issue or infrastructure problem. We do not need to consider the issue of software or hardware when we see these. For software, we have our own set of return codes that return a code and user-friendly description of the issue. The code itself is used as a key to lookup what the exact problem is (class, method, problem), etcetera. Let us say that we have a microservice that reads a user profile. Perhaps the profile cannot be found. Instead of returning a 404, making us go through a bunch of tasks to determine if the issue is that the request URL is wrong, the database has no records, or something is invalid in the reverse proxy or enterprise caching; return a code that means that the profile is not found! For example, if the profile is not found return 1001 with a description such as user profile not available.

Speeding up troubleshooting

To troubleshoot this issue, we already know that the problem is not infrastructure because we did not encounter a 404, 500, etcetera. We know that the problem is software. We go to a document that takes that code and tells us where the problem is. Then through analytics, we track the codes by functions over time to look for user experience anomaly trends. If we see 500, 1001 codes in our analytics, which we know are the user profiles not being found, we know that, for some reason, the record is not in the database for these users. We do not go down a rabbit hole trying to differentiate between software and infrastructure, and we isolate the problem almost instantly without a large stack trace snapshot continually occurring, slowing down the application.

Perhaps we have another call across the network to a service to submit information for processing and an authentication request for someone logging into the platform that leverages an LDAP service to do so and, in the same request, also grabs some feature authorization information from the application database. Do we send back a 401? No, if the LDAP technology has an issue, it may throw a 401 and then we know exactly where the issue is. For the software itself, we pass back a return code for that function. Perhaps that code is 1002 returned with a description of not being able to authorize the user. In this case, we may look at 1002 and look up documentation that says the user was not in the database in this table. We know that the software itself is encountered, and we have granular, detailed information about what the issue is. The goal of good error handling is to not just say that the application had an issue but to accurately identify exactly where that issue is as far as root cause without any ambiguity. The goal should always be to not need 10 people to triage an issue trying to find root cause. The software should be smart enough to know what happened. It knows what it tried to do and failed at. Throwing a 100 like stack dump is not helpful in resolving the issue quickly.

Make sure that we use an error code and throw exceptions that identify exactly where the issue is (class, method, source file, specific problem and include data that encountered the condition). Be wary of putting **personally identifiable information** (**PII**) data in a log. This is generally a no. We want to obfuscate the data and use IDs in table lookups instead to represent where the record is indirectly. Some organizations also leverage hashing, but we recommend using a method that does not indicate personal information in the logs.

```
try {
  // Code that may throw different types of exceptions
  int result = divideByZeroErrorExampleAction();
  System.out.println("Divide Result: " + result);
} catch (ArithmeticException | NullPointerException e) {
  // Handle either ArithmeticException or NullPointerException
  System.err.println("Error 1000 occurred in service, in this class… when
  executing this method... when doing this operation with this data...: " +
  e.getMessage());
} catch (Exception e) {
```

```
  // Handle generic unexpected type of exception
  System.err.println("General Exception Error 9999: " + e.getMessage());
}
```

When it comes to response codes, always return a message or number than means something. Do not rely on backend HTTP standards, let those reflect what is happening from a hardware and system perspective. For the software that we have control over, return error numbers that represent success or specific error numbers that can be used to look up the issue in detail. This will save us a lot of time in troubleshooting an IT platform. We should not be returning 404s for database lookups when information is not found, no matter how many blogs are read on the internet, saying this is the standard way of handling errors for micro-services or RESTful services. Instead, we have a list of error codes to pull out in the response that have meaning directing us very granularly to the exact problem that is occurring.

If the error is going back to the user, then we represent an error number and a user-friendly message such as the system is busy, please try again later. Perhaps we execute a transaction that inserts data into the database. The data, however, is bad data that will not be inserted due to the database's unique key constraints. In that case, the error message should indicate which data is bad and tell them to resubmit or retake the action. The error number associated with the message should be a key to be used by IT support personnel to know the class, method, and action to take to address the issue and gather relevant user impacting error metrics over time.

To reiterate, all errors logged on the backend or front-end UI console should indicate exactly what happened, exactly where it happened in the source code, the data involved in the error and an error number that can be used by anyone to determine what to do about it by looking it up in a support document or knowledge share. This is what allows a third party to support the application or another team who has limited knowledge of the IT product. We do not just print a stack trace, which is very impactful to performance and is not clear as to the exact issue without a lot of tracing through it which is time consuming. If we have a service that inserts into the database and the operation fails due to the field width not being long enough in the database, indicate a return code for the transaction that occurred that indicates the method operation failed during insert, show the class, method, source file, data, etcetera. Unless there is an error an HTTP 200 should always be returned for expected behavior.

Just by showing this code, we should know the SQL that was executed that failed. For the reason why the database did not like the SQL, we throw up an exception in this case with logging with more detail indicating the specific SQL error that occurred and when throwing up the exception, we include source file, class, method, error response, data (SQL) and have a parameter that indicates if the error is critical. Only exceptions that show up as critical should force a page and immediate action from support personnel with a page. But remember, if we write to a log, the result should be something actionable. This means the issue should be fixed and the error message should no longer be in the logs. Logs should

Now we know that with microservices, it can be popular to use HTTP codes for return codes for services. This is not ideal; if we do end up leveraging this type of approach for some reason, make sure that all alerting and error checking is through exceptions and a combination of return codes and not just the return codes themselves. We need to know exactly where the issue is and if it is our code or code in infrastructure. The best approach is to just use a HTTP 200 for success and anything other than that is a failure. So, if we are doing a database lookup and a record is not found and this is normal behavior, return a 200 and not something like a 404. If it is not normal behavior, ideally, return an error code like we discussed earlier; if we do return a 404 and we decide to map return codes to software functions that we write, then at least we should have an exception that indicates the details around that 404.

Java exception types

There are two categories of exceptions in Java, **Checked Exceptions** and **Unchecked Exceptions**. Checked Exceptions are conditions that an application should anticipate and throw up the call stack and recover from them. These are always checked during compile time to make sure the necessary dependencies exist. Unchecked Exceptions, are unexpected run-time exceptions that we were unable to predict. These are not checked during the compile process.

For best performance, we want to throw Checked Exceptions as early in the code block as possible. Then we want to leverage the finally block to clean up any database connection references, set large objects to null for garbage collector cleanup, close streams, files, etcetera. Another option is to use try with resources to auto close resources. The finally block always executes no matter what Java exception occurs. But remember, exceptions are a last resort. Creating an exception object means allocation of memory and initialization which is slow when the object contains an embedded stack trace and needs to transverse the call stack. Some important items to note when it comes to checked exceptions:

- Only use exceptions for exceptional conditions, not for program flow. Make this a part of ALL code reviews.
- Check arguments coming into a method for validity at the beginning of the method—This allows us to fail quickly when the parameters for a method are not valid versus causing havoc throughout the rest of the method causing failure after failure.
- Use custom error codes as discussed instead of relying on exceptions whenever possible. These custom errors are critical to getting to root cause of an issue and at the same time preventing instantiation of large exception objects with large call stacks to an exception. When an exception is thrown, the JVM has to transverse the run-time call stack, which is expensive from a performance perspective. Exceptions are a last resort because they significantly degrade an applications performance. Custom error codes should be leveraged for services versus http codes and also for internal operations in the application where needed. This stems from older

- We write unit tests that validate the correctness of our error checking. This is crucial by the way.
- Profile code to determine where exceptions are impacting performance the most while running unit tests validating our test logic. We want to optimize code to minimize exceptions, throw exceptions early in methods for best performance.

If we are leveraging an abstraction framework like Spring, we can setup our error handling using custom errors:

- Create a global handler for exceptions using the **`@ControllerAdvice`** annotation to define a global exception handler.
- Create a custom error response class with details such as custom status code, error message with source file, class, method, data, and timestamp. This class would then be returned by the exception handler.
- Implement different methods that handle exceptions with an annotation of **`@ExceptionHandler`**.

Remember though that an abstraction framework transversing the Java call stack during run-time like this using a large abstraction framework is very slow. We do not want exceptions to happen except in very rare cases beyond our control that are actionable, instead we want to leverage custom error codes taking us right to the issue in the name of performance and resolving issues quickly. Raw stack traces are slow, degrade our platforms performance significantly and ambiguity around error code sources, where the error really occurred, should not exist. We need to be very detailed. The application knows itself why it broke. We need to make sure our ITM platforms are good communicators.

JavaScript exception types

JavaScript is a computer language heavily used on the front end for web applications. uses a single type of exception called **error**. However, just like Java developers can take the same approach. Create different types of errors in JavaScript with error codes that correspond to different issues. Like Java, JavaScript leverages try/catch/finally to handle exceptions, separating code into manageable segments for effective error scrutiny. There are several types of errors we can throw which indicate a different kind of problem. But just like Java, exception throwing should be rare and actionable and not something that happens except in exceptional situations. This assures good performance because JavaScript, just like the Java programming language, encounters the same latency challenges. Console logs in browsers should always be clean and not cluttered with exception after exception. Leverage error responses and only throw exceptions when warranted. They should be very rare. The normal approach from a performance and MTTR perspective is to leverage custom error responses to represent an exact issue and make the code fast without large stack snapshots that need to be transversed by the run-time environment and reach out to many IT support staff members for troubleshooting.

Log standardization and documentation

We have talked a lot about exception handling, error codes and not logging large call stacks into logs. But let us talk more about logging. When it comes to logging, first we need to choose a fast-logging framework. Java does have logging built in and Log4J is popular, but what we see used most these days is Logback and SLF4J. However, keep in mind that aggregate logging frameworks that are abstractions on top of lower-level logging frameworks offer flexibility around plugging in new code but hurt performance. Here are some key points that we want to consider when implementing logging.

- Use a standard logging format.
- Use log levels.
- Log levels should be switchable without a new release of the software (configuration).
- Aggregate and centralize logs and have them visible as single a pane of glass dashboard, as we talked about early in the chapter.

  Note: We need to make sure that we include the source so that we can drill down further.

- Only setup alerting on fatal/critical errors for support.

The different log levels are:

- **Debug**: We modify the configuration of our application to reflect debug when we want detailed level tracing. This should be done rarely as it impacts performance. If we are looking for detailed performance traces, we leverage OpenTelemetry or **application performance monitoring** (**APM**) tools such as Datadog, Dynatrace, Application Dynamics, etcetera. We can also leverage tracing APIs in cloud providers but generally OpenTelemetry is an abstraction layer on top of these with a standard. Many profilers like Java Flight Recorder are also beneficial.
- **Info**: This is not verbose logging or used for application tracing. It is used to indicate that components are starting up, configured a certain way, more of a simple operational or configuration status at run-time.
- **Warn**: Non critical issue or not sure if it is an issue type of alert.
- **Error**: Run-time error—not necessarily something to page a support person on.
- **Fatal/Critical**: These alerts indicate a pageable issue. No other log exceptions should cause a pageable support condition.
- **Trace**: Line level tracing versus higher level debug tracing.
- **Off**: Disable all logging.

When it comes to log generation, we cannot stress enough how important log documentation is. Where we assign an error number with descriptions of issues where the error exists, such

beneficial as far as cost, when a third party is involved for IT platform support or there is turnover in the organization with important intellectual capital on how the platform works.

Here is why we care about log documentation:

- **Reduced MTTR**: Detailed information can guide anyone, whether it be the support team or the development team, to the root cause of the issue quickly, with no bloated call stacks necessary.
- **Great to hand support to other teams and resources**: Detailed documentation supports third-party teams in understanding our logs and addressing issues minimizing impact to other teams doing delivery.
- **Over time it continues to evolve for faster resolution of issues**: Log documentation with action points can be used and updated as issues are learned, evolving resolution times to be even faster in the future as the knowledge base grows.

Conclusion

In conclusion, we talked a great deal about exceptions and logging with respect to performance. How throwing exceptions should only be done in exceptional cases and leveraging custom error codes offered significant advantages for the performance and support of an IT platform.

Verbose logging or poor exception handling and frequent exceptions being thrown can introduce considerable performance issues. Performance is not the only issue as far as the application, it can also be a factor in the performance of the team resolving issues. When logs are too bloated with noise, sifting through them to locate what is really a problem becomes difficult task and a time consuming one. The result is slowing down the IT response to customer issues and potential revenue.

Poorly handled exceptions when they are thrown up a call stack will often result in an unstable platform or even crashes. Performance testing of platforms with a large number of exceptions can be difficult. It is hard to truly reflect the correct scale. Performance testing is only valid if the test data reflects the exact same workload model with exceptions thrown. So, if exceptions are many and often, the testing results are completely invalid unless the test data and scenarios invoke the same exceptions at the same frequency.

Balancing the right amount of logging and implementing efficient error handling strategies is crucial to ensure optimum performance and system stability. Efficient logging involves logging just the right amount of data at the right time. For error handling, adequate testing, understanding of system dependencies, and handling exceptions gracefully with the needed cleanup and ensure a resilient, high-performing application. There is no denying the importance of effective error handling and logging. They not only help maintain the health and performance of our applications but also offer valuable insights into user behavior and system interactions. In the next chapter, we will go deeper into cloud observability and profiling.

Chapter 12
Cloud Observability and Cloud Profiling

Introduction

This chapter of the book will talk about observability in the cloud and profiling code in the cloud. In order to optimize software and find the root causes of bottlenecks, it is necessary to have good observability into the depths of our IT platform. We cannot put enough emphasis on the idea that this is not optional and it should be done in lower development environments early on. We have seen some teams that will develop an entire application without observability in place, seeing it as a secondary component or afterthought until the platform goes live to support it in production. The result is that teams spend weeks trying to solve anomalies during the development process using trial and error approaches with code which could have been solved in minutes if observability had been put in place early.

Structure

This chapter covers the following the following topics:

- Introduction to observability
- JMX and JTI
- Observability performance impacts

Objectives

In the last chapter, we talked a great deal about logging and exception handling and how this impacts performance. We talked about common sense approaches to error handling for IT support staff and messaging to users of the platform. In this chapter, we are going to focus entirely on observability on both the frontend and backend and see how we can understand what is going on from a performance perspective.

We will talk about the different cloud observability solutions that are native to each vendor and go deeper into how to do thread and stack dumps to find contention and how to do heap dumps to see what classes are on the heap consuming memory.

Introduction to observability

Observability in cloud computing gathers insights samples of our platform metrics, looking for data patterns, trends, and insights into anomalies. It exists not only on the backend of an IT platform, but end to end from the user's browser all the way to a backend web service. It must, not only trace the entire end-to-end traffic patterns across network switches, balancers, reverse proxies and JVMs, but also dig deep into the run-time environment itself and profile the code and run-time utilization dynamics. In most cases, it also means understanding log analytics and what is being written to logs over time and the state of the platform.

Many times, observability means multiple tools, perhaps tools such as a code profiler, an observability tool like OpenTelemetry, a packet analyzer like Wireshark and many times APM tools such as Extrahop or Datadog. Even perhaps another solution may be leveraged to see deep into the database like *Oracle AWR reports* or *Google Spanner Query Insights*. No matter if we have found an all-in-one solution or individual tools, it is important to have a single pane of glass as we discussed in the last chapter, in the interest of finding the root cause quickly when performance issues arise. We want high level observability, that can quickly drill into the other solutions easily when needed to go deep into the run-time, database, code, etcetera.

Why observability matters

Imagine trying to fix a television without being able to see under the hood. Sounds impossible, right? This analogy holds true when it comes to managing IT platforms in the cloud without observability. If we cannot see what is happening in our run-time environment, we will waste an enormous amount of time guessing what the issue might be through code changes and redeployments. Leverage solutions like OpenTelemetry for tracing and some deep profiling tools like JProfiler, **Java Flight Recorder** (**JFR**), cloud profilers like the GCP profiler, etc. can help you optimize your platform. The most comprehensive solutions are feature rich all in one **application performance monitoring** (**APM**) solutions that can provide the needed visibility into traffic patterns, responses, and with deep analysis of run-time environments and the

down to the line of code causing the issue and see the correlation between the run-time environment metrics and that line of code.

How to find which tool to use

Picking tools in Gartner's magic quadrant over time is a good approach to finding out which observability tools have the best feature set over time and what companies are leveraging. Something to watch out for is agent file sizes being deployed to containers that consume too many resources and any latency that may be introduced when incorporating the solution. Some commercial observability solutions are really old legacy applications catered towards on-premise solutions that have been retrofitted for the cloud. These tend to have fat agents that consume resources on VMs, Kubernetes pods, etc. In this case, we lose scale through observability and run-time costs increase. But remember, observability is not optional. We cannot turn it off and do without it. Balancing run-time costs with feature sets and agent sizes is a good thing to do.

Each cloud provider has their own observability solution. Some of these are better than others. We will talk more about these in this chapter. We need to be able to know how many threads are running in our JVM, the line of code that is causing latency, and how the call being made has contributed to latency end to end along the path to get to the destination service or application. We need to be able to see not only inside a VM or a Kubernetes pod, but be able to differentiate between heap and off heap memory being consumed. It is important to have visibility into objects on the heap and see thread times in wait conditions that may be contributing to wall time, which is the time elapsed from when we first ran a program or service to when it finished.

We need to be able to see CPU time and how much time the CPU is spending processing requests. If we are using web services with Tomcat, we need to be able to see how many connections are coming into our service and how many threads each connection is spinning up in the JVM, how many requests are getting queued on the network connection. Another valuable insight that is to be able to see when the **just-in-time** (**JIT**) is running. Java applications can easily get into constant recompiles and code cache flushes when configured improperly for the application running. Observability at great depth across our run-time and network will save us much time and cost and our customers will be happy because our **Mean Time to Repair** (**MTTR**) will go down.

What cloud providers offer for observability

In this chapter, we will explore observability for three of the most popular cloud platforms, **Google Cloud Platform** (**GCP**), AWS, and Azure. We will dig deeper into the robust monitoring tools offered by GCP, AWS, and Azure, and learn how they can help us optimize the performance of our cloud applications. Each of the major cloud platform providers, including GCP, AWS, and Azure, offer built-in observability and logging tools that work seamlessly within their environments; though be sure to note that most of these are not as

feature rich as third-party solutions to identify the root cause around performance issues quickly.

- GCP leverages a tool called an **operations suite** (formally **Stackdriver**) as a logging solution and monitoring. It provides detailed logging info and detailed metrics. We can easily query logs in GCP and see metrics around restarts, CPU and memory utilization. At a detailed level, to see beyond logs and deeper into the run-time environment, we need to install the GCP profiler which is recommended. We only need to do this if we are not leveraging JFR directly or running another run-time profiler (JProfiler, etc.). It is also possible to leverage OpenTelemetry with GCP that uses the Cloud Trace API that *Google* provides under the covers. We can take that data and push it into Prometheus and Monarch and store the data over multiple years.

  One of the lightest weight solutions for monitoring that is multi-cloud and can be used for on-premise that scales well is Prometheus. We would highly recommend that we always think in terms of portability around multi-cloud solutions in anything that we design or develop, it will save us cost down the road. When it comes to monitoring, consider a solution that supports multiple clouds and on-premise.

  For Prometheus, we can even use Grafana, the industry's popular dashboarding tool with the powerful query language to retrieve metrics from an Influx DB time series database to store data over many years of time instead of GCP's Monarch storage, if wanting to have a vendor independent solution. One thing to consider with any monitoring solution is that we want at least one year's worth of data to understand behavioral and seasonal patterns. Many APM solutions collect too much data and can only store data for a few weeks or it becomes no longer cost effective. Instead, we can use Prometheus with a data storage solution such as Monarch, which, at the time of this book writing, allows for 2 years of storage for free. Then use Grafana for our representation. Below is documentation around GCP monitoring. Think in terms of needing information for troubleshooting at the deepest level but for longer term storage higher level capacity related metrics. More information is as follows:

  - **https://cloud.google.com/monitoring/docs**
  - **https://cloud.google.com/stackdriver/docs/managed-prometheus**
  - **https://cloud.google.com/stackdriver/docs/managed-prometheus/query**

- AWS provides CloudWatch and X-Ray. **CloudWatch** monitors resources used by applications in AWS, on-premises and can even monitor other clouds. It can respond to performance changes, optimize resource use and provide insights into operational health. **X-Ray** is a full APM service that collects data about requests that our application serves and provides detailed metric insights that we can use to identify and address performance

tracing and debugging. AWS also supports Prometheus and can use PromQL to filter and query time series information across hundreds of Kubernetes clusters. Like many Prometheus managed service cloud solutions by the major providers, it automatically scales as query needs increase on the platform. Below is documentation around AWS monitoring.

  - **https://aws.amazon.com/cloudwatch/**
  - **https://aws.amazon.com/xray/**
  - **https://docs.aws.amazon.com/prometheus/**

- *Microsoft Azure* features a tool referred to as **Azure Monitor**. This is an APM service like CloudWatch that can be used to monitor live applications, identify performance anomalies, and diagnose issues with detailed tracing. Azure performance solution is impressive with its visual representations showing dependencies and component interactions like some of the more expensive APM tools in the market. It uses a feature called **Application Insights**. Like all providers, Azure also supports Prometheus as an easy-to-use service. Here is more information on using Prometheus with Azure. Most clouds provide an all-in-one solution with Grafana and long-term storage around performance data.

  - **https://azure.microsoft.com/en-us/updates/general-availability-azure-monitor-managed-service-for-prometheus/**

All of these tools have decent visibility into their environment, with Azure having the best easy-to-use solution, in this author's opinion, followed by AWS and then GCP, which is more limited. That being said, *Google* is really making strides in artificial intelligence, so expect this to get more integrated and visual representations to get better over time. But to date, the others have better monitoring solutions. To have visibility into our run-time environment and code, without an APM solution with built-in profiling abilities, we need to first activate a profiler in GCP. Activating the *Google Cloud Profiler* on our GCP infrastructure is a straightforward process. It only requires a few steps:

- **Select profiler**: Navigate to the *Google Cloud Console*, and on the navigation menu, click on **profiler**.
- **Configure application to support Profiler**: We need to include the cloud profiler agent in our application's build. Update all code dependencies and ensure that the profiler agent is initialized at application startup.
- **Cloud profiler agent**: In order for the profiler to know what to monitor, pass in the service name and service version to the profiler. This is the unique identifier for our service. The same service name should be used across all instances. The version is something helps us identify different deployments of the service for monitoring.
- **Running the application**: When we run the application or service, the profiler agent will automatically scale and begin

from our application and run-time environment. It is important to give the agent appropriate permissions to generate and view the profiling data.

Once we have successfully completed these steps, the Google Cloud Profiler will be enabled on our application running in the GCP infrastructure. We can then use this tool to analyze CPU usage and memory utilization, latency data, and understand the performance of our IT platform in real-time. This information is important to improve our application's performance and help us make more strategic decisions for further optimizations.

As we use the profiler, we will learn how important metrics like CPU time, heap memory utilization and wall time identify our bottlenecks. Knowing how to enable and effectively use Google Cloud Profiler is an essential part of maximizing the performance of our application running on the GCP platform. If we are looking at performance, the first metric to consider is wall time for threads and methods and then drill into details from there. But one of the most often things overlooked is JIT recompilation. Many profilers can identify when this occurs through a flame graph identifying bottlenecks or specific metrics being tracked (Datadog, etc). We want to be able to determine what percentage of the time our code is performing recompilation and optimize the run-time environment to minimize this to improve our performance.

AWS is another cloud provider that is popular in the industry and is one of the most mature cloud platforms; it has been around much longer than the rest and to date still has the most market share. It provides many services designed to help businesses scale and grow and rapidly create infrastructure, software or leverage on demand managed services. It excels in providing large compute capacity faster and more affordably than traditional on-premises servers like many providers. It can, in many instances, be more expensive than Azure and GCP, but it is also more capable than the others; though Azure has made some great strides to steal market share. GCP was late in the game compared to *Microsoft* and *Amazon* and is in catch up compared to these other two. It is slowly becoming an impressive solution for business. But do not count it out, *Google* was slow to enter the phone market and the *Google Pixel Pro* is one of the most intelligent phones out there with its seamless integration with the *Google infrastructure*.

When it comes to *Amazon*, the king of the three cloud providers, AWS X-Ray is the observability and debugging tool available that provides deeper insights into performance trace metrics. This solution gives us good insights into the behavior of our applications, helping us optimize the platform and assure optimal performance. It is a decent tool that assists in improving the understanding of how our application and services are performing. We can even have open-source standard tracing tools that support more than the JVM, like OpenTelemetry, which sends information to X-Ray, which provides detailed level tracing and debugging for applications at a granular level. Turning on X-Ray involves some straightforward steps:

- **Install X-Ray SDK**: We need to integrate the AWS X-Ray SDK with our application or service in order to enable detailed tracing. AWS supports a variety of languages such as Golang, Java, Node.js, and C++.

- **Instrument**: After installing the SDK, we must instrument our application by importing the SDK classes, initializing a global tracing client, and configuring the service map.
- **Rules**: X-Ray uses sampling algorithms to determine which requests to trace. The rules can be customized for different services.
- **Enable**: The X-Ray daemon must be enabled and running so that it can gather raw data and relay it to the monitoring visual interface.

Enabling AWS X-Ray for our applications can be a large step towards identifying performance bottlenecks on AWS and assuring an optimal user experience.

The last cloud provider that we are going to discuss is Azure. Like AWS and GCP, Azure is capable of delivering numerous SaaS, PaaS, and IaaS services using large data centers. It has seamless integration with other *Microsoft* products including IDEs as we would expect. **Azure Monitor** is Azure's built in observability tool which leverages something called **Application Insights** to gather deep levels of metrics. It provides a good solution for collecting, analyzing, and reacting to raw data and feedback from our cloud applications. It helps us understand how our applications are performing and proactively identifies issues affecting them and the resources they depend on.

To enable Application Insights in *Microsoft Azure*:

- **Enable deep application monitoring with Application Insights**: This is the tool within Azure Monitor that will provide us deep application diagnostics and performance monitoring.
- **Configure**: Configure Application Insights to gather deep performance metrics and diagnostic information. Set up alerts, queries, and analytics to capture specific application events or trends.
- **Azure Monitor dashboard**: The Azure Monitor dashboard will populate with data gathered by Application Insights. We will be able to response times, server requests, exception rates, and other important performance metrics. Moreover, Azure Monitor offers a feature called **Application Map** that gives us a visual overview of our application's components and their interdependencies. This is one of the most important features of any observability tool, to be able to see a big picture diagram around traffic being sent to and from components with request sizes and latencies. It tells us what is talking to what.

Gaining valuable insights into our application's performance and uncovering potential problems is one of the essential aspects of cloud profiling. Azure Monitor helps us achieve this by offering a range of profiling and performance metrics such as request rates, response times, failure rates, and resource utilization, to name a few. It also can provide analytics of not only the server but browser page performance as well so we can see what a user is experiencing.

JMX and JTI

Beyond what the cloud providers provide internally, many times it just comes down to JMX arguments or programming to the JVM's tool interface to see the level of detail needed and this can be the lowest cost solution to run and implement. The **Java Tools Interface** (**JTI**) has been pivotal in providing a means of communication between Java applications and various programming tools. With this interface, tools can control not only the execution of code but have the ability to access very detailed run-time information to inspect a programs state and debug complex applications. Developers have the ability to achieve real time debugging running either locally or remotely.

But there is another interface that has been around longer called the **Java Management Extensions** (**JMX**) interface. With this interface, we can quickly monitor our platforms health, determine uptime, resource usage, GC information and thread dynamics. Many **application performance monitoring** (**APM**) tools leverage this under the covers, but it is easy to use without a third party tool using simple socket programming or a basic command line tool like curl. All we need to do is find the MBean that has the attribute we are interested in. Then we just open a terminal window in our operating system and use a command like:

```
curl: http://[host]:[port]/jmx?qry=[MBean name]
```

It is that easy, just make sure that the port is open through any firewalls that may be in place. Many times, we want a simple report of the state of our Java run-time environment. We can easily grab the values from JMX and convert those values into HTML using a JSON processor like jq or other.

jq is popular because it is a JSON processor that has zero run-time dependencies written in portable C language and has a great deal of output filtering available.

https://jqlang.github.io/jq/

Here are where we can find filters for various formatting conversions:

https://jqlang.github.io/jq/

We can also pipe output to tools like awk and other Linux tools for manipulating the output to create very advanced HTML presentations.

When it comes to free monitoring, we can achieve some terrific metrics between JMX, JTI, and tools like JFR and **Java Mission Control** (**JMC**). Combine OpenTelemetry for tracing using the cloud trace APIs under the covers and we have some powerful support capabilities with little cost.

Java Mission Control

JMC is a set of powerful tools that are a part of the *Oracle JDK*. They are used for managing, monitoring, profiling and troubleshooting a Java application. JMC is used with JFR to

analysis. These tools can gather information while an application is running minimal impact on their performance. They are designed to run in production with no issues at all. Using both tools together can provide detailed diagnostics and profiling with JMC providing nice visual representations of the information with detailed tables and analytical charts. What is unique about JMC is its ability to provide detailed diagnostics and profiling of Java applications, including those that are running in production. It offers visual interfaces with detailed system charts and tables, allowing developers to understand the behavior of their applications incrementally and spot inefficiencies or issues.

OpenTelemetry

OpenTelemetry tooling is all about tracing to give a 360 view of performance and is great for operational high level observability across layers in an infrastructure. It is more about creating an open protocol solution that multiple vendors can adhere to in order to achieve full tracing end to end including outside and inside of the JVM. This has been a gap for larger organizations for a long time. Today, balancers like *F5* support OpenTelemetry, reverse proxies and web servers like Nginx support Open Telemetry and other vendors have been heading that direction as well. This is the value of OpenTelemetry. It is for high level tracing across the entire infrastructure. There is still a long way to go, but this simple tracing protocol that can be used across balancers, reverse proxies, JVMs, adds quite a bit of high level performance value that was unable to be achieved easily before. It can support both frontend and backend traces.

When it comes to drilling deeper, OpenTelemetry is not always practical to use. We would not want development teams trying to continually re-release/rebuild their tools to provide deeper tracing or try to trace at a deep level out of the gate. It would not be efficient. Instead, deeper diving requires much more coding/tracing at a more granular level or integration using profiling solutions that not only monitor the code, but the run-time environment running the code to correlate inefficiencies and bottlenecks.

This data-driven perspective enables us to make informed decisions that can boost the performance and reliability of our applications. OpenTelemetry is not a replacement for APM or run-time profiling tools, it is more catered towards end-to-end tracing at a high level and is a terrific open solution for creating high level operational dashboards that incorporate the full path through the network versus being limited just the JVM. Standards like OpenTelemetry have opened the gate for vendors of all types to support tracing that was not there before. We can now trace through balancers and proxies at a granular level and see latency end to end as it goes across the network.

Of course, when it comes to the JVM, one can continue to release software with more granular OpenTelemetry traces to drill further as a solution, but it is generally not practical to trace at the line level. Profilers or even better, all in one APM monitoring solutions that can deep drive into the run-time environment and allow the line of code granularity provide significant value for deep dive troubleshooting and intelligent trend tracking/alerting.

These solutions work great. But sometimes, we may want a more catered solution that meets our specific implementation. When it comes to Kubernetes monitoring, the most popular tools out there are Sysdig monitoring and of course tools like Prometheus that track time series events over time. As we discussed previously, most cloud providers have a Prometheus service that can be used for monitoring and feedback scaling. These solutions or other similar solutions that monitor and feed data into a time series database like Influx DB or that use Grafana for dashboarding are very flexible cloud independent solutions.

Observability performance impacts

What we will find when looking at solutions built initially for on-premise solutions versus cloud solutions is that these solutions can sometimes be to heavy weight for the cloud. For example, if we install a large APM tool, something like Dynatrace, Datadog, Application Dynamics, etc., generally agent file sizes exceed ¼ of a gigabyte in size.

If we have Pods running Docker with JVMs in Kubernetes that leverage 512Mbytes for our services, then adding in a large agent for each Pod that consumes a great deal of disk, memory and CPU resources can be costly to run over time. But tools like Prometheus or SignalFx are almost 1/3 of the size of these other solutions and much lighter weight. Solutions built for the cloud versus retrofitted for the cloud are almost always lighter.

Advantages of expensive APM tools over free options

When it comes to the costly commercial tooling around observability it is all about reducing MTTR. Many of these tools have artificial intelligence or modeling pattern recognition to capture when things are going awry quickly and identifying early when our customers are experiencing an issue. Many times, this comes in the form of frustration indexes or other proprietary metrics. In other words, the value of costly commercial tools is not just in data collection and observability as much as it is about data interpretation and trend detection.

What do APM tools do

Essentially, APM tools are software designed to monitor the performance of our software applications, identify potential issues and bottlenecks very quickly to find root cause, and provide actionable insights for optimization. Many have excellent alerting capabilities that cater to customized needs and metrics. They can be vital for keeping applications running smoothly and efficiently, especially in large-scale environments such as cloud-based platforms. These have evolved as all-in-one solutions that provide a combination of tooling around logs, profiling and even enablement of the JFR or JMX attributes. Many can do thread stack dumps and heap analysis as well.

They try to cater themselves as a one-stop-shop for performance, but we find that many times the network appliances, reverse proxies, balancers, API gateways, and network

like OpenTelemetry that can be used across devices. It is hard to completely get to one single tool due to the variety of network appliances out there. No doubt, at some point in the future, cloud providers may put all of these other solutions out of business, but that time has not yet come. At the time of this book writing, Datadog and Dynatrace are considered the top APM solutions. Both have the ability to deep profile to the code level though profiling solutions and find bottlenecks.

The primary objective of APM tools is to be able to support detailed code and JVM profiling, observability, and intelligence to drill into root causes very quickly. Most do this very well and provide metrics like a frustration index when response time thresholds are exceeded for requests and the user is having a bad experience. This is the level at which the user will bounce off the platform, out of a flow, or start hating the company brand. It captures the detailed performance data, analyzes it, and then provides visualizations or summaries that make it easier to understand what is happening within our application. This performance monitoring, in turn, greatly contributes to overall cloud observability.

Supporting the IT platform

A reinforced understanding of the application's performance history, error logging, and user transactions can provide invaluable insights into the system. When something goes wrong in the complex world of cloud computing, APM tools can help us track down the issue, often straight to the line of code. Most have tried to take market share away from log analytic solutions by providing their own query languages and log analysis themselves and correlating it with performance bottlenecks. This correlation can be powerful when it comes to resolving time sensitive revenue impacting incidents quickly.

Metrics such as the number of threads going into service, threads used in the JVM, and other metrics can be tracked and analyzed with these tools. This detailed information can be used to correlate application metrics with JVM metrics, providing a comprehensive view of how our application's overall performance is impacting the user experience. To use JFR, all that is required is to enable it when the JVM starts using the **`-XX:StartFlightRecording`** argument. Once JFR is enabled, it will start collecting profiling data that we can analyze using the JMC tool.

Understanding thread dumps

Almost all cloud environments support running the JVM today as the software run-time environment, including GCP, Azure and AWS. The JVM run-time environment will many times leverage threads to execute operations in parallel like we have discussed in the past. This allows the JVM to handle multiple units of work simultaneously. But sometimes, threads can have contention on a shared resource which makes them block and bottleneck the application.

To determine where this is happening in the code, we can generate a thread dump from the Java run-time environment that will tell us the state of each thread. This is done in many ways, such as using a **`Kill -3`** statement on the PID for the OS running on the virtual machine or using **jstack,** which is a command line

run-time environment. Many times even observability tools like Dynatrace (APM tool) will call these commands under the covers. To get the PID in Linux, we just type the **`jps`** command at the command line and find the PID for the process we are interested in. Then we just execute something like:

```
jstack PID > dump.txt
```

We can also easily get the objects in the run-time environments heap memory using a simple statement like:

```
jmap -histo PID
```

Many versions of the JDK come with a tool called **jconsole**. With this, we can retrieve detailed thread statistics for the run-time environment, which can be essentially for troubleshooting. If we are doing a lot of memory dumps and thread dumps, we recommend leveraging the Eclipse **Memory Analyzer Tool** (**MAT**) or VisualVM; these are great tools to visually see what is going on in the JVM. Remember that more threads being used, does not necessarily translate to better performance. We can run across something called **thread thrashing** or **excessive context switching** where the platform spends more resources managing threads than executing the application itself.

We have talked a great deal about the run-time environment, observability and profiling on the backend, but remember that there are some great tools on the frontend using Chrome developers tools. With this, we can profile our JavaScript and run something called **Performance Insights,** which will tell us the web core vitals for a web page. Another great tool of course is Lighthouse for scoring how well our application is coded for performance.

Analyzing code upfront

Code analysis tools like PMD and SonarCube can provide some good code related metrics and help us optimize the solution. Chrome and also many other browsers support the ability to see the waterfall chart showcasing artifacts being downloaded and rendered in the browser while providing key insights around blocking, queuing and latency around requests. Also remember that even tools like webpack used for packing JavaScript has analysis tools and measures we can take to achieve great performance. Not to mention frameworks like AngularJS.

Conclusion

In conclusion, we talked a great deal about observability and profiling in the different cloud environments, along with some of the other tools that provide value for troubleshooting difficult performance problems. We talked about JMX and JTI and how to achieve great observability with free tools. That being said, we also discussed more comprehensive, but expensive solutions like APM tools.

In *Chapter 13, Alerting Strategies*, we will go one step further and put it all together. We will talk about alerting strategies and how to put a great strategy in place to eliminate the noise

Chapter 13
Alerting Strategies

Introduction

In the last chapter, we talked about observability in the cloud and profiling code in the cloud. In this chapter, we are going to focus on creating a good alerting strategy. Why do we care about alerting? We care about alerting because it is crucial to reduce how long customers are impacted when there is a problem in production; it is important to improve the time to acknowledge an issue and the time it takes to repair the issue, which we have talked about in previous chapters. Sometimes, this is referred to as **mean time to acknowledge** (**MTTA**).

A good alerting strategy means minimization of false positives which is probably the greatest challenge that we have encountered. When support personnel are continually paged with alerts for items that are not actionable or that indicate a problem when a problem does not truly exist, those pages quickly start being ignored. When this happens, customers suffer. When software is being deployed into production, we need to have a mechanism to disable the alerts and then re-enable the alerts after the software is released. If we do not, it means pages out to people wasting their time thinking that an issue is occurring when it is not.

We have seen some teams that will develop an entire application and put it into production without any alerting whatsoever in place. It is important that alerting be put in place before software is released into production as part of the checks/balances in the release process. In other words, there should always be a list of

before releasing software, and alerts are one of the most important considerations for support. These are identical steps that should be taken for all software that is being put into production. There is nothing worse than having our customers find a problem before we do or having alerts in some areas of the application but not in others because alerts were not considered top of mind.

If we are not alerting on anomalies in production, then there is no use having observability. Observability is in place so that we can have alerting to tell us when something is slow or when something has gone wrong in the infrastructure or software impacting the customer.

Structure

This chapter covers the following the following topics:

- Introduction to alerting
- Setting up alerts in the cloud
- Paging hierarchy/support processes

Objectives

In the last chapter, we talked a great deal about observability and profiling. In this chapter, we are going to focus on implementing a good alerting strategy. Previously, in this book, we talked about logging and how important it was to only log actional items. If something is being written to the log that is not actionable, then we should not be logging it. This ties into alerting. We do not want false positives with alerting due to poor logging standards and we do not want to rely on high level return codes for services or high-level application modules only without considering the anomalies that may be occurring underneath.

We talked about common sense approaches to error handling for IT support staff, determining the root cause of issues when something goes awry, leveraging observability and profiling. But how do we know when to alert and what to alert? In this chapter, we will talk more about alerting in general and discuss these topics along with different cloud vendor solutions around alerting.

Introduction to alerting

Receiving information about what is happening on the platform at a detailed level when an unexpected event occurs can save countless hours in addressing an issue. Alerting is an important component of system monitoring. It will help to identify root causes around issues quickly when implemented correctly. Many organizations leverage multiple cloud technologies in their end state solutions, including when operating across internal workgroups. Setting up a successful alerting strategy across multiple cloud platforms like **Google Cloud Platform** (**GCP**), Azure, and **Amazon Web Services** (**AWS**) requires a tactical approach.

We have seen common logging frameworks like Splunk used for alerts for organizations supporting multiple cloud and on-premise solutions. Standardizing actionable logging is crucial for this to work well, but it does work well when implemented properly. A centralized alerting strategy does not always need to be logging, it can be a common tool like an APM tool as well.

In this chapter, we are about to discuss how to implement an effective alerting strategy that helps maintain availability and system resilience. But remember that the strategy is not about alerting for everything and anything that comes up, but instead, determining what actions require immediate attention to take action upon to minimize the impact on revenue and customers. What alerts have enough severity where support needs to be immediate. These types of alerts should not be frequent, or often; if they are, there are bigger organizational problems to solve around the IT platform or product creation process.

Let us go deeper and provide some insights into setting up alerting for GCP, Azure, and AWS, considerations for a strategy that keeps the noise of alerts to a minimum, and the importance of resilience, performance, and availability metrics to monitor. We will also review paging hierarchies and support processes. As an IT professional, alerting can be a complex topic, there is something here for everyone.

The primary focus will be on the following items:

- Deriving a noiseless alerting strategy
- Focusing on resilience and the right metrics to alert on
- Paging hierarchies
- Alerting in Azure, AWS and GCP
- Support processes/alert management

To derive a noiseless alerting strategy means to focus on when an anomaly occurs that is a high enough priority to immediately address. This is an alert that is revenue impacting, worth getting up 10 people in the middle of the night to address and not wait until the next morning. First let us remember that we do not alert on every error that shows up in a log or based on a single return code that comes back negative from a service. We do not alert on entry validations that fail or even data coming into a service that is failing a schema level validation where the data is not of the same type, length, etc. We do not wake people up in the middle of the night for a single synthetic failure. We wake people up in the middle of the night when we see a failure or trends impacting revenue.

Large organizations typically play it safe and adopt hybrid or multi-cloud strategies using the GCP, *Microsoft Azure*, and AWS. If we are not setting up a central alerting mechanism through centralized logging or a centralized APM solution, it means that we are typically setting up alerts in these different cloud providers, from OpenTelemetry traces and a combination of APM and observability tools.

Most APM tools support OpenTelemetry out of the box for alerting and can alert on custom metrics or formulas we set up. This can be extremely powerful to use as an alerting approach that can be leveraged across both multiple clouds and on-premise solutions. Setting up an efficient alerting system is important to ensure uptime and consistent service performance. If we are leveraging multiple clouds and also on-premise, we highly recommend that we leverage APM solutions and solutions supporting the OpenTelemetry protocol for alerting across the entire technology stack being used. APM or common logging solutions are the common denominator that can be leveraged across multiple solutions. The last thing we want to do is have different alerting approaches for GCP, Azure, AWS, and on-premise solutions. We want the same alerting strategy for our entire enterprise; we want to standardize monitoring and alerting.

As far as when to alert, we alert on actionable failures impacting revenue and trending changes that impact the user experience, including performance. We take multiple measurements without a timeframe before initiating any alert. For example, if a 10:00 am on Wednesday during the week, we expect to see 200ms for a service on average and we see 500ms, and we see that for three measurements over a 15-minute window, this would be a good condition for an alert.

It is important to understand the behavior of our application for metrics as well over time and the standard deviation around each metric. So, for performance, a good alerting threshold is to add the largest 5 latencies observed over 90 days for a service. Then, add the highest 95 peak average response time over the past 90 days. Add these numbers together and divide by 100. This is a weighted average threshold that does a great job of catching issues without causing noise around alerts.

Most cloud providers have the concept of **service level objectives** (**SLO**) these days. An SLO is essentially a target level of service that generally includes response time, error rate, and availability but it can be more or less metrics depending on the depth considered. It is a set of criteria that the service promises to meet a certain percentage of the time. The purpose of an SLO is to set clear and realistic expectations, these can provide crucial insight into the service reliability and user satisfaction.

Many APM tools allow us to define custom metrics to trace SLOs by or cater them towards response time as a starting point. Cloud providers generally allow us to define alerts based on these metrics. For example, if our service is supposed to have a latency of no more than two seconds 95% of the time, that's the SLO. Once our SLOs are defined, we are ready to start setting up alerts.

Setting up alerts in the cloud

To set up alerts on GCP:

To start, we will need to navigate to our *Google Cloud Console* and access the Cloud Monitoring service. As a precursor to defining alerts, we want to identify the key metrics

for our microservice. For instance, we might be looking at latency, error rates, CPU or other factors depending on our SLO. Then, we want to set up our alerting policies based on these metrics. To do this, go to the GCP Console and click on **Monitoring**. Then, navigate to alerting. Once we are there, click **Create policy**.

Now we need to set up our condition. To do this, we select the type of resource we want to track and define our threshold condition SLO and frequency of when to alert; for example, 3 measurements over a defined SLO in a 15-minute window is a good approach to noiseless alerting. Then we decide on our notification approach for our alert such as SMS, application, email, or integrations with chatting tools like Slack or paging solutions like PagerDuty. Remember that we need to save our policy and test our alert to make sure it meets the necessary criteria.

One important aspect of setting up alerting strategies and SLO thresholds is that these periodically need to be reviewed and adjusted over time as the platform evolves. Many times, for example, an application needs to respond to the user in 4 seconds, and it makes 5 micro-service calls to retrieve information. We may decide to have one micro-service be faster and do less work and another do more work over time to achieve that same 4 second latency. As the software changes, the granular SLOs much change as well.

Now let us look at *Amazon's* solution and learn how to set up alerts on *Amazon CloudWatch* when we see an SLO exceeded. SLOs are not just about failures, it also can mean that our platform no longer has the capacity it should have and can quickly become overloaded. When response times go high, capacity can be quickly impacted. So, when we see values go from 2 second response time to 8 second response times, it may be an emergency because the platform cannot handle the upcoming load during the peak hour of the day beyond the poor user experience being seen. We need to leverage SLOs to make sure that the service quality remains consistent but also help us avoid alert overload. We have talked about noise a great deal in this chapter. Do not over-alert and have a strategy on what to alert on and what should be considered an emergency.

Remember to alert that:

- Similar to logging, all alerts should be actionable and warranted
- Alerts should indicate the root cause or point out exactly what is occurring in detail
- Alerts should not be noisy

Setting up alerts is crucial to ensure our system's availability and performance. But we do not want to overwhelm our team or our team's responsiveness will decrease. For the *Amazon Cloud* instead of the terminology for alerting, we will see the terminology set as alarms. To get started setting up alarms in AWS, we need to access the *Amazon CloudWatch console*, select **Alarms**; then select **Create alarm**. In the **Select metric** area, we select the specific metric we want to monitor. After selecting our metric, we will then define the conditions for our alarm/alert.

We will find that *Amazon* has much more capability than the other clouds when it comes to what an alarm can do and complexity around metrics, including initiating EC2 scaling activities. Once we are done defining our alarm, we just add a name and a description, and then the alarm is set.

The largest competition to *Amazon* is not GCP; it is the *Microsoft Azure platform,* which offers a great deal of alerting options to make sure the platform performs and is available. Similar to AWS, Azure provides customized alerts that have complexity and multiple dimensions to ensure our services are functioning optimally. Regarding Azure, alerting is easy to use, like most *Microsoft* products. We setup alerts through the alerts service in the Azure portal.

Azure has a powerful alerting strategy like *Amazon* and has built in a great deal of features to look for relative changes. At the end of the day, we are looking for changes in our environment that is unexpected regardless of the metric. If we see something is relatively changing, we are detecting a trend changing; this can give us a head start on addressing the issue before it becomes a full outage. Azure's solution involves real-time alerting on both metrics and logs and also something called **smart alerts** that are powered by machine learning. Like the other providers, we can set up alert rules for efficient alerting. Alerts in Azure can be set up to cover more than just set thresholds being exceeded that many tools have.

Like some of the models and plugins available for Splunk, a popular log analysis solution looking for pattern changes, we can set alerts for anomalies, spikes, or falls in trends, relative changes, etc. Unlike some of the other providers, Azure alerts can act as an early warning system that helps prevent system failures, performance degradation, and other disruptions to our applications running in the Azure cloud. Splunk also has similar models that can be embedded in the solution when a centralized logging strategy is defined. It can use models to detect pattern changes to catch issues in a platform before something occurs that impacts revenue or the customer. Whether we are tracking failure rates, latency, trend changes, getting alerted immediately when something goes awry is critical. Alerts in Azure can be delivered through the Azure portal, email, webhooks, SMS, communications apps such as *Slack* or *Microsoft Teams*.

To setup an alert in Azure, open the Azure portal and navigate to the **Alerts** section. Click on the **Monitor** tab that is shown in the left-hand menu. Select **Alerts** and then click the **new alert rule** button and define the details around the alert. We will see a **Target** field that will allow us specify the resource that the alert should monitor. Then click on the **Condition field** to define the condition. For example, did the CPU exceed 90% for 10 minutes time. The finally, like the other tools, configure our notifications, SMS, PagerDuty, Webhooks, etc. and save our policy. Like mentioned before, we always test our alert!

Not only is it crucial to not generate noise around alerts, it can be also beneficial to define severity, if we have alerts that are not severity 1 alerts that wake people up in the middle of the night for something impacting revenue. For these alerts, which may be sent via email

or less critical timed messages, always specify a severity in the message when we can, along with a great deal of detail to allow us to get to the root cause quickly. Something also to consider is to first pool our alerts and send them out in one message over time for less severe alerts. For example, send a list of alerts in one notification vs. one per notification. If you do not, you can quickly fill up someone's email or put them in a recurring paging cycle.

When defining alerts, alerts should be geared around **key performance indicators** (**KPIs**) and availability metrics, which typically means considering metrics such as latency, error rates, and system resource utilization when configuring alerts. We want to build our alert strategy over time and cater towards only actionable alerts. The same goes for improving our logging. We tune our alerting strategy to not waste our team's time and resources.

Part of an effective alerting strategy is having a reliable paging system in place. Pagers are essential tools for communicating critical information quickly, and tools like PagerDuty can be good solutions for managing pages and hierarchies. Always have a robust support process to respond to paging incidents. Who is on call that will be alerted, who is the backup, who is the escalation point, and at what interval does each page occur? Have the ability to cater notifications to people with expertise around the user journey in the IT platform to resolve the issue. Fast, quality and clear communication leads to a faster response, which ultimately reduces the impact on service availability and system performance.

Paging hierarchy/Support processes

Optimizing our infrastructure for prompt alerts is not complete without an effective support process. When an alert is triggered, reaching the right person in the team swiftly is paramount. Good alert management signifies building comprehensive paging processes that align with our team's capacities and response readiness.

Most cloud service providers like GCP, Azure and AWS, offer native solutions to help manage paging. The alert should contain comprehensive data about the anomaly encountered, aiding in a faster resolution. Remember that the software should know what is wrong with it, build in that intelligence, and page out exactly what the issue is, service level, class level, method level, and exception type (custom if it can further refine the specific issue as part of the notification). Title the alert with the workflow impacted at a minimum and the severity. Do not get lazy and send a stack trace; if we can do that and we have proper error checking, we know what happened, and we should communicate what happened. Try not to make people drill into a stack trace from an alert, say what the problem is, and identify where it is being seen. We know generating large stack traces is the most expensive operation we can do when it comes to performance.

Paging hierarchies' direct alerts to the right individuals or groups and dictate the flow of alerts in an organization. An alert might initially go to a junior support element. If the issue is not resolved within a designated period, the alert may then escalate to a more experienced team member. To build an effective alerting hierarchy, it is important to define

our team's responsibilities clearly. Understand expertise and where escalations should route. It may not be as easy as using organizational hierarchy.

It is important to ensure the resiliency of our alerting infrastructure. This means having redundancy. We do not want notifications and alerting to have a single point of failure where alerts do not get triggered and support people do not get notified. We want to make sure that we also have a mechanism to easily disable alerts during software deployments and releases to keep people from being notified that a problem is occurring when it is actually not occurring.

Conclusion

In conclusion, we talked a great deal about alerting, how the different cloud providers set up alerts and notifications and establishing a support process hierarchy that does not always align with an organizational hierarchy but instead by expertise on the system. More than anything else, there was a lot of emphasis on not setting up noisy alerts and setting up only actionable alerts with clear information as to where the issue is occurring and what the issue is to allow the root cause to be defined quickly and clearly.

In *Chapter 14*, *Future Trends in Performance Engineering,* we will go into the future and talk about future trends in performance engineering. The next chapter will be an exciting one that showcases where the industry is headed.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 14
Future Trends in Performance Engineering

Introduction

This chapter of the book will talk about future trends in performance engineering and where this is all headed. **artificial intelligence** (**AI**), advances in web technologies around fetch priorities, pushing more towards fetch API solutions over XHR requests, and finding better and faster ways of executing quality gating both functional and performance through automation with CI/CD will become prevalent. This chapter will discuss future trends and deeper integration of AI and machine learning algorithms in performance monitoring tools. Imagine automated systems that not only detect anomalies but also predict potential issues before they even occur. Performance engineering will increasingly rely on distributed monitoring solutions that can handle the complexities of multi-cloud environments and edge resources. These platforms will not only integrate seamlessly with existing systems but also offer comprehensive, end-to-end visibility into cloud performance.

As these trends continue to evolve, staying ahead of the curve will be essential. Aligning an organization with next-generation monitoring solutions will be crucial in maintaining not just the performance but also the competitive edge of IT infrastructure in a fast-moving digital landscape.

Structure

This chapter covers the following topics:

- Future predictions
- Fetch priority API
- Performance automation with behavior driven development
- Artificial intelligence

Objectives

In the last chapter, we talked a great deal about alerting building upon previous chapters on logging and observability/profiling. In this chapter, we are talking about the future and how IT platforms are evolving. In this chapter, we are going to focus entirely on the future of both front end and backend technologies.

We will talk about AI and how design patterns and frameworks are evolving. The fetch priority API will be discussed along with new approaches to automation in performance engineering practices.

Future predictions

The pattern of frontend frameworks operating locally on the browser with desktop like features reaching out to backend microservices with isolated datastores or SOA services with common datastores and efficient localized batch processes will continue as a design pattern for IT organizations. More pervasive computing techniques, web worker thread management, async programming, event driven design solutions, self-contained microservices, and continuous delivery with blue/green deployments will continue to evolve with IT platforms supporting front end HTML5 features that all browsers support.

Device farms

Using large device farms in testing like LambdaTest or BrowserStack is becoming mainstream to make sure that features work with responsive web design and different devices and browsers. Using the performance APIs in parallel with device farms adds detailed frontend performance metrics into the load testing and analysis equation. These farms provide a great deal of coverage around testing both from a functional and performance testing perspective.

Frontend performance and optimization

Frontend web performance and optimization will become more important than ever before

where **content delivery networks** (**CDN**) become more feature rich with more availability around the world. Browser caching techniques and staying away from ETags (except for content management platforms), and moving towards longer term caching with hash naming to change hash names on releases will become prevalent to eliminate requests across the wire, further improving capacity and response times. Chaos engineering will continue to evolve, and vetting SLOs while chaos engineering takes place to ensure SLOs are still met will continue to evolve. For example, degrading the platform by 20% and running load tests and functional tests to see how the platform behaves and if it self-heals or has the necessary resiliency measures such as failovers and traffic routing patterns, timeouts, etc.

Improvement in delivery processes

We will see more emphasis on direct to production solutions and quality gating with continual delivery over lower environment heavyweight processes. This will further evolve towards blue/green or direct to production from development saving cost. NoSQL solutions will continue to provide the flexibility needed for cloud web applications, eliminating the parsing and object mapping conversion needed across distributed platforms, instead supporting JSON, which the browser supports out of the gate. Grabbing data directly from the database to the browser without mapping latency in the different tiers; however, it is clear that the durability and data integrity of relational solutions have not been outmatched by NoSQL when it comes to billing events or important payment data where integrity, durability, and consistency are most important. Organizations continue to rely on these robust and feature rich relational systems, with *Oracle*, MySQL, SQL Server, PostgreSQL being the top 4 databases used throughout the world.

NoSQL

MongoDB is currently the only NoSQL solution that has gained comparable market share to relational systems for real time systems and it falls in right behind PostgreSQL as far as popularity and usage. Where NoSQL solutions are prevalent, beyond large distributed data farms, is when it comes to observability with time series databases such as InfluxDB and Prometheus, with InfluxDB having far more market share than all of the others (Graphite, Kdb, TimescaleDB, DolphinDB).

The reason MongoDb is far ahead of the other NoSQL solutions for real time solutions is that it is an exceptional document NoSQL database that is flexible and very fast. It can operate with available memory, keeping a lot of records readily available, and it is written in C++ vs. slower languages requiring an intermediary runtime environment. It also has enterprise level security with seamless encryption/decryption, making it fast when working with sensitive data. In other words, it is enterprise grade representing real time capabilities that can keep up with the best relational system with multi-tenant security features built-in. It can be used across clouds and on-premise, which makes it a flexible solution without needing to work about working across

cloud scenario. The ease of use and robustness compared to solutions written from open-source toolsets wired together or slower virtual machine languages have made it very successful. This is what organizations like Panera use when we order our latte from their ecommerce application. It is a much more solid solution than the solutions that most cloud providers provide.

Frontend web frameworks

Also, front end web frameworks like React, Svelte, and Vue will continue to evolve, with more bloated frameworks like Angular losing more market share over time as trends continue. React is currently in the top spot as far as the mainstream web framework at the time of this book's writing. We want a framework that is fast, easy to use, and that supports responsive web design across browsers and devices for web applications. Being lightweight and fast should be a top consideration over bloated features we may never use.

Solutions with **Virtual Document Object Models** (**DOMs**), which leverage in memory representations of the real browser DOM, have a significant performance boost over alternatives. But any web framework that is lightweight and takes performance seriously is becoming popular. It is good to see all of this emphasis on performance. Right now, web frameworks Solid, Qwik and Svelte are the top new frameworks developers are interested in. For the backend, we see RUST and Golang being popular solutions for web services and batch programming with respect to great performance. Python is still popular for data mining, but far too slow for real time services. It excels in expression matching well but is extremely slow beyond that compared to other languages. NodeJS with JavaScript is very fast as long as no intensive computing or parsing is taking place. This solution works great when paired up with a document datastore like MongoDB.

Better native compilation solutions

When it comes to Java, solutions like native Spring using GraalVM is starting to evolve. We see a lot of projects by *Google* and other cloud providers looking to make Spring boot native a reality. Spring has become a large fat library that is slow to load today in Java runtime environments; going native significantly improves startup times in pods in Kubernetes. This means handling surge traffic better with fewer performance outliers. Quarkus services are an alternative to Spring with even faster startup, but they do not have enough market share compared to Spring to achieve the needed traction.

Self-healing

Over time, the cloud will become more advanced in how scaling and observability happens and how it self-heals and protects itself. There is no doubt that observability across cloud and on-premise hybrid solutions will evolve, and aggregation around views

models will further increase. So, we will be able to have multiple metrics and custom metrics monitored over time with standard deviations recorded around metrics and be alerted when behaviors drive outside of expected metrics and behaviors within a defined time window.

Managed services

The same goes with evolving services; we see better and better managed services in cloud environments, which make everything a black box. This, in-turn causes vendor lock-in but organizations are quick to see cloud applications as something that can be recreated quickly if needed. For large enterprise solutions, we recommend sticking with more cross cloud databases (MongoDB, PostgreSQL, etc.) and cross cloud solutions like Kubernetes and abstraction layers over these for portability. For smaller applications without sensitive information, it is less important.

Virtual IDE solutions

We will start to see more remote desktop and virtual build environments occurring where everything is in the cloud versus using the bloated, more difficult IDEs that are local to the desktop. Local environment setups are time consuming for new resources starting, and it is difficult to maintain consistent versioning. Cloud IDEs are the future and are where things are headed. What we see still on-premise today as far as IDEs, will start evolving to the cloud more over time.

Fetch priority API

One of the newer APIs that is evolving is the fetch priority API. Early in this book, we talked about how a browser parses a web page, and when it does this, it finds images, scripts, CSS, etc. When the browser does this, it automatically assigns them a fetch priority that we have no control over. It tries to determine the best order and retrieves the resources based on this priority setting.

But to give us even more flexibility, now browsers are starting to support the fetch priority API and a fetch priority HTML attribute. This can help us optimize Core Web Vitals to improve the responsiveness of our web applications. For example, we can improve **Largest Contentful Paint** (**LCP**) metric dynamics by setting an image with an HTML attribute such as **`fetchpriority="high"`** on the image element, causing LCP to happen sooner. We can also increase the priority of async scripts and decrease the priority of scripts farther down the page in a similar fashion, how about that for power. This flexibility gives us many tuning and optimization options for page load and interaction times.

The fetch priority API determines the order in which network requests are made based on their importance. It categorizes requests into different priority levels like low, high, medium, etc. This differentiation allows more critical resources to load first, drastically

improving the user experience for page interaction. For example, crucial elements like webpage content and images might be labeled as high priority, while less essential components like analytics scripts could be allocated a lower priority.

When we make a request using the fetch priority API, it returns a JavaScript promise that resolves once the fetch operation is complete. The final output is a response object representing the response of the request. This object contains information about the response, including the status code, status message, headers, and the data itself.

This fundamental understanding of how the API works will help us better assess the performance of different software architectures, as it directly reflects their ability to effectively manage and prioritize network resources.

Performance automation with behavior driven development

One of the challenges of performance testing is many times, we have fast release cycles, and creating large complex performance scripts slows down release cycles. To alleviate this problem, tests can be easily incorporated into functional testing. This is done by leveraging toolsets like Cucumber, JMH or JMeter. Then, easily generate performance tests using Gherkin and **behavior-driven development** (**BDD**). For example, in Cucumber we may have a feature file with a scenario just like any other functional scenario, except for performance. It is important to learn Cucumber when it comes to automated testing; we can plug it in with tools like REST assured for API testing or `HTTPClient` classes built into Java.

https://cucumber.io/

Cucumber feature file:

A feature file in Cucumber serves as a blueprint for test scenarios and is what allows for more BDD approaches to testing. One of the key advantages of using a feature file is that it bridges the communication gap between technical and non-technical stakeholders. Since feature files are written in plain English, everyone from developers to product architects can understand and contribute to the development process. This collaborative approach ensures that all aspects of the application requirements are covered and correctly understood. It minimizes requirements and misunderstandings around translation.

The human-readable format that Cucumber uses is called **Gherkin syntax**. Gherkin is a simple computer language that is easy to learn, with no IT programming training required, which details the specific functionalities and behaviors of an application. This Gherkin syntax includes reserved keywords like *Given*, *When*, *Then*, *And*, and *But* to define the prerequisites, actions, and outcomes of your tests.

Feature files promote test automation and continuous integration. Once the scenarios in a feature file are defined, they can be linked with

execution of tests during each build process. The tests can be run for both performance and functional testing as part of the quality gating process, and the build can fail if quality gates do not pass.

Another advantage is maintainability. As applications evolve, feature files can be easily updated to reflect new features, changes, or enhancements. This makes it simpler to manage and adapt your test cases in alignment with the project's progression. Here is an example of a feature file:

```
Feature: Performance Testing

  Scenario: Measure Response Time Under Load
    Given the system is under heavy concurrency
    When I send X number of requests concurrently
    Then the average response time should be below Y milliseconds
```

To implement steps to support the Gherkin scenario above, we create something like the following:

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;

public class PerformanceTestStepDefinition {

    @Given("Given the system is under heavy concurrency")
    public void givenSystemIsReady() {
        // Setup actions
    }

    @When("{int} When I send X number of requests concurrently ")
    public void whenUsersAccessApplication(int numberOfUsers) {
        // Use JMeter or JMH to simulate user load with the specified
        // number of users
        // We may call JMeter/JMH APIs here to execute the load test
    }

    @Then("Then the average response time should be below Y milliseconds")
    public void thenResponseTimeWithinRequirements() {
        // Perform assertions on response times, possibly using JMeter
results
    }
}
```

JMeter example:

Let us provide a solution that shows the implementation of using both JMH and JMeter. One of the most important advantages of using the JMeter jar with Java for performance

testing is the flexibility it offers. By integrating directly with your codebase, you can customize and automate tests to suit your specific needs rather than being constrained by the limitations of a standalone tool. This gives you the power to create more precise, targeted tests that closely mirror real-world usage scenarios.

Another benefit is the seamless integration with existing DevOps workflows. By embedding performance tests into your CI/CD pipeline along with quality gating to ensure that every build is vetted for performance issues before it reaches production. This proactive approach helps identify and resolve potential bottlenecks early in the development cycle, improving application reliability and user experience. The ability to leverage Java extensive standard libraries and third-party integrations can significantly enhance your testing capabilities. For instance, you can easily incorporate data from various sources, like databases and REST APIs, to build comprehensive, multi-faceted test scenarios and test strategies from data creation to storing and replaying, along with unlimited potential to do anything you want. APM and other monitoring tools that offer integration support with Java can be leveraged to collect and analyze monitoring data, providing valuable insights that help optimize cloud performance in real time. Here is an example of an implementation for a feature file written in Cucumber that leverages an implementation in Java, leveraging JMeter in this instance:

```
import org.apache.jmeter.engine.StandardJMeterEngine;
import org.apache.jmeter.reporters.ResultCollector;
import org.apache.jmeter.save.SaveService;
import org.apache.jmeter.testelement.TestPlan;
import org.apache.jmeter.util.JMeterUtils;

public class JMeterExecutor {

    public static void executeTestPlan(String testPlanPath) throws
Exception {
        // Initialize JMeter
        JMeterUtils.loadJMeterProperties("path/to/our/jmeter.properties");
        JMeterUtils.initLocale();

        // Load test plan
        SaveService.loadProperties();
        TestPlan testPlan = SaveService.loadTree(new File(testPlanPath));

        // Run the test plan
        StandardJMeterEngine jmeter = new StandardJMeterEngine();
        jmeter.configure(testPlan);
        ResultCollector listener = new ResultCollector();
        jmeter.addTestListener(listener);
        jmeter.run();
    }
}
```

JMH example:

When combined with a frontend HTTP call or lightweight web framework to hit a backend service, JMH can become a full-blown load testing tool with much more accuracy than JMeter. One of the key advantages is that JMH provides precise and reliable benchmarks, ensuring your performance tests are accurate to the point of contributing no more than 2ms itself, unlike JMeter, which is around 100x greater. Java developers can measure the performance of their code at a very granular level, making it easier to identify bottlenecks and optimize critical sections of the application.

Moreover, JMH simplifies the whole benchmarking process. We can rely on JMH's robust libraries to handle the intricacies of warm-up phases, measurement iterations, and simple execution. This built-in functionality reduces the time and effort required to set up and run performance tests.

Another notable benefit is JMH's ability to generate visual detailed reports over time. These reports highlight important metrics like throughput, latency, and execution time. With this data, developers can easily track performance improvements or regressions over time. Like JMeter, using JMH and coding in Java directly allows for unlimited flexibility. Here is an example of implementation for a Cucumber feature file written in Java with JMH:

```
import org.openjdk.jmh.annotations.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Benchmark)
public class MyBenchmark {

    @Benchmark
    public void measureResponseTime() {
        // Implement the logic to measure response time here
    }
}

To Run Tests

import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/resources/features", glue = "path.
to.our.step.definitions")
public class CucumberJMeterRunner {
    // This class serves as a runner for our Cucumber tests
```

```
}
```

We can also leverage testing frameworks like RestAssured in combination with JMH to have it be a full load testing tool for APIs such as:

```
import io.restassured.RestAssured;
import io.restassured.response.Response;
import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@Warmup(iterations = 10, time = 10)
@Measurement(iterations = 10, time = 10)
@Fork(2)
@State(Scope.Benchmark)
public class RestAssuredBenchmark {

    private static final String OUR_URL = "https://api.ourapi.com";
    private static final String ENDPOINT = "/our/api/endpoint";

    @Setup
    public void setup() {
        // Setup configurations for RestAssured
        RestAssured.baseURI = OUR_URL;
    }

    @Benchmark
    public void restAssuredAPICall() {
        // Make the API request using RestAssured
        Response response = RestAssured.given()
                .when()
                .get(ENDPOINT);

        // Optionally, perform assertions on the response if needed
        response.then().statusCode(200);
    }

    public static void main(String[] args) throws Exception {
        org.openjdk.jmh.Main.main(args);
    }
}
```

Artificial intelligence

In the future of performance engineering, AI will play an important role in changing the way we optimize infrastructure and address performance issues. With the right observability and feedback loops, and traffic modeling, we can self-tune our infrastructure and have it evolve over time. Remember that AI, combined with observability, can easily identify root causes when it identifies SLOs being breached with good accuracy. But it is not just about diagnosing an issue, it is about offering a solution and acting upon that solution in real time. Imagine an AI that could regularly run safe tests in production for performance, analyze capacity, self-tune, then monitor traffic in real time and standard deviations across metrics and self-adjust for optimal resource usage and performance.

AI algorithms cannot only monitor traffic patterns, but they can also leverage machine learning to predict future performance issues to prevent downtime and impacts on the user's experience. Imagine having a service be able to direct traffic to an operating pod, change a configuration and automatically restart/refresh with a new web service configuration and continually learn.

Perhaps the platform detects a performance delay in a web service. It first identifies that the problem is a slow query; it then augments the query and tests the latency in a controlled sandbox environment; once ready, it augments the code to include the new query, compiles, rebuilds, and deploys. It finds the most effective solution is implemented to restore optimal performance in both code and infrastructure by taking a traffic pattern optimizing and vetting in a controlled environment before implementing it all in real time. Perhaps we design the application with the necessary separation of concerns to allow this type of control easily.

AI could also not only look at code, infrastructure considerations but also be able to see dependencies between services and even recommend optimizations around architecture. It may even be able to replace layers of architecture. The possibilities are endless. What about performance testing?

An AI could automatically write performance automation scripts or code for new features implemented for each release. So manual scripting would no longer be necessary; it could do performance testing in a production sandbox before allowing a service to be deployed into production as part of the quality gating process. What about functional testing?

Automation around functional testing, reading code, coming up with unit tests, integration tests, and negative and positive scenarios is all within the reach of AI, and then setting up quality gates around releases into production as part of the build process. Even when it comes to traffic balancing. AI could facilitate automated system load balancing based on anticipated demand, reallocating resources as needed or scaling outward/inward according to the anticipated changes in load or traffic.

In the short term, AI such as ChatGPT will allow us to automate coding, testing, setup boiler plate performance tests in JMH, JMeter, Cucumber, REST assured, and API testing

frameworks like Karate. We will soon see AI that can completely code, test, and release software. Today, with ChatGPT, we can create test plans, do code reviews, write code, perform unit tests, etc., all leveraging AI. We just need to ask the right questions in the right amount of detail. We can even use it to generate test data!

https://chat.openai.com/

Conclusion

In conclusion, we talked a great deal about the future of performance engineering, possibilities around improving Core Web Vitals for a web page using the new fetch priority API. We talked about trends in web frameworks and how AI will soon predict behaviors and help us self-tune our infrastructure, code, and testing/quality gates. We talked a lot about merging performance with BDD and creating automation as features are developed for an application.

We sincerely hope that this book provided some useful information along with way. Performance engineering is an art more than anything else that includes software, infrastructure, architecture, databases, and networks. We will not find a more fulfilling career than to pursue learning about all aspects of tuning, analyzing, coding, and designing software for great performance! This expertise will evolve over time along with the paired practices around chaos engineering. By understanding these two disciplines, we can really ensure that we have an outstanding high-performance, resilient platform.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Index

Symbols

<HEAD> 127
<script> 178
-g:none 188, 189
-Xcomp 184

A

Agile Manifesto 53-56
Agile Manifesto,
    core value 53
Agile Manifesto,
    principles 53, 54
Ahead-Of-Time
    (AOT) 186
Alerting 294-296
AOT, benefits
  CPU Utilization, reducing 186
  performance, optimizing 187
  startup speed, increasing 186
AOT, drawbacks
  limit, optimization 187
  platform, independence 187
AOT, steps 187
API Performance,
    improving 267
API Performance, strategies
  IP/URL 268
  lease, connecting 268
  least, time 268
  round robin 268
  sticky round robin 268
  weight round robin 268
API Resource,
    fetching 305, 306
ArrayList 220, 224
Artificial Intelligence (AI) 311
Asynchronous
    Programming 217-219

AWS X-Ray 286
AWS X-Ray, steps
  enable 287
  instrument 287
  Rules 287
  X-Ray SDK, installing 286

B

Batch/Real-time,
    separating 262, 263
BDD 306
BDD, architecture 306-309
BDD, steps 307

C

Caching 140, 141
Caching, algorithms 142
Capacity Planning 112-114
Capacity Planning,
    best practices 125-130
Capacity Planning, concepts
  distribute platform,
    scaling 119-121
  environment, testing 115
  optimization,
    scenarios 114
  percentiles,
    configuring 116, 117
  short-term events,
    interruption 117, 118
  software capacity 121, 122
  third-party,
    dependencies 123
  traffic pattern,
    correlation 114, 115
Capacity Testing, insights
  resource, utilizing 64
  scalability 64
  thresholds 64
CDN, key points
  content retrievals,
    performing 168
  HTTP Protocols 171
  Lossy, optimizing 165, 166
  multi-layer, caching 167
Chaos Engineering 106, 107
Chaos Engineering,
    architecture 108
Chaos Engineering, principles
  failover, systems 107
  fault tolerance 106
  redundancy 106
Chrome Developer 155
Chrome Developer,
    capabilities 155
Cloud Performance 87-89
Cloud Performance,
    challenges
  Cloud Services, managing 90
  impact, performance 90, 91
  region, latency 92
  scale impacts, resilience 92
  security, impacts 93
Cloud Performance,
    measuring 94, 95
Cloud Platform 85, 86
Cloud Platform, advantages
  Collaboration, efficiency 87
  Cost-Effective 87
  Disaster Recovery 87
  scalability 86
  software, updating 87
CMS Collector 195
CMS Collector, capacity
  heap size, optimizing 196
  JVM, optimizing 196

CMS Collector, drawbacks 195
CompletableFuture 217
Content Delivery Network
(CDN) 165
CPU Impact 220
CPU Time 93

D

DevOps 56

E

EDA, characteristics
asynchronous 132
limit, polling 132
real-time 132
EDA, key points
asynchronous,
communicating 103
scalability, elevating 103
system responsiveness,
improving 103
EJB, advantages
downstream systems 228
performance 228
reliability 228
EJB, disadvantages 227
EJB, strategies
First-in First-Out
(FIFO) 228
Least Frequently Used
(LFU) 228
Least Recently Used
(LRU) 228
Most Recently Used
(MRU) 228
Random Replacement
(RR) 229
Enterprise Java Beans
(EJB) 227
Error Handling 273
Error Handling,
opportunities
actionable, logging 273
IT Platform, supporting 274
Java Exception 277, 278
JavaScript 278
Java With Exception,
handling 274
troubleshoot,
speeding up 275-277
Event-Driven Architectures
(EDA) 102

F

Functional/Non-Functional,
balancing 16

G

Gand Of Four (GOF) 229
GCP, setting up 296-299
GOF Behavior, patterns
command 231
interpreter 231
iterator 231
mediator 231
memento 231
observer 231
responsibility 231
state 231
strategy 231
template 231
visitor 231
GOF, patterns
abstract factory 230
builder 230
factory method 230
prototype 230

GOF Structural, patterns
adapter 230
bridge 230
composite 230
decorator 230
facadade 230
flyweight 230
proxy 230
Google Cloud Platform (GCP) 294

H

HashMap 221

I

IFrame 176
IFrame, creating 178, 179
IFrame, disadvantage 177, 178
IFrame, tricks
caches, harnessing 176
headers, configuring 176
HTTP Request, minimizing 176
lazy, loading 176
iron triangle 45-47
IT Agile 52
IT Agile, impacts
Agile Manifesto 53
DevOps 56
Project Delivery Resources 57
IT Iterative 51, 52
IT Performance Cycle 61
IT Performance Cycle, concepts
API Level, testing 63
capacity, testing 64
code, analysis 64
design, considering 65-67
Driven Design/Development 62
infrastructure, analyzing 65
Journey Level, testing 63
IT Waterfall 48
IT Waterfall, phases 48
IT Waterfall, steps 49, 50

J

Java Mission Control (JMC) 288
java.util.HashMap 221
java.util.Stack 223
java.util.Vector 222
Java Virtual Machine (JVM) 182
Jenkins 146
Jenkins, key points
JMH 150, 151
LambdaTest 151
MTTR/MTTA 148
SLA 148
SLI 148
SLO 149
Jenkins, tools
Continuous Delivery 147
Continuous Integration 146
Performance, testing 147
quality, gating 147
JIT Compiler 199
JIT Compiler, techniques
profile-guided, optimizing 199
startup time, optimizing 199
threshold 199
JVM, benefits
code, integrating 186
less, memory 185
performance, improving 185
platform, implementing 186

JVM Code, key points
code, synchronizing 213, 214
efficient data, structure 212
exception,
handling 215, 216
native, compilation 209-212
queuing 215
threshold, compiling 208, 209
JVM Code, optimizing 206-208
JVM, considerations
applications,
precompiling 183
code cache, optimizing 183
Garbage Collection,
optimizing 184
native java,
applications 182, 183
JVM, de-compilers 185

K

Kubernetes 134
Kubernetes, aspects
autoscaling 134
secret/configuration,
managing 135
self-healing 135
service discovery/load,
balancing 135
storage, orchestration 135
Kubernetes, strategies
partitioning 138
read replicas 138
sharding 138

L

Log Documentation 279
Log Documentation,
key points 279
Log Documentation, levels
Debug 279
Error 279
Fatal/Critical 279
info 279
off 279
Trace 279
Warn 279
Logging 272

M

Martin Fowler, principles 233
Micro-Services 103
Micro-Services, architecture 104
Micro-Services, benefits
fault, isolation 103
flexibility 103
scalability 103
Micro-Services, navigating 105
Micro-Services, rules
independent 132
scalable 132
specialized 132
Microsoft Azure 287
Microsoft Azure, insights 287
MTTR/MTTA 148

N

Network Tuning 124
Normalization/Denormalization,
comparing 240, 241
NoSQL Databases 242
NoSQL Databases, aspects
compression 245
fast writes 245
scalable 245
time oriented 245
time series, functions 245

NoSQL Databases, attributes
concurrency 244
flexibility 244
scalability 244
write, speed 244
NoSQL Databases, benefits
performance 243
scalability 243
NoSQL Databases, fundamentals
performance 244
replication 244
scalability 244
schema 244
NoSQL Databases Scaling, types
Horizontal 242
Vertical 242
NoSQL Databases, types 243

O

Observability 282
Observability, architecture 283-285
Observability, impacts
APM, optimizing 290, 291
code, analyzing 292
IT Platform, supporting 291
MTTR, identifying 290
thread dumps 291, 292
Observability, importance 282
Observability, steps 285
Observability, uses 283
Off-Heap Storage 197
Off-Heap Storage, entities 198
Off-Heap Storage, factors
code cache, controlling 198
concerns, separation 198
heap space, maximizing 198
On-Premises Platforms 84
On-Premises Platforms, advantages
internet, dependency 85
physical security 85
On-Premises Platforms, concepts
customization 85
quality, assurance 85
response time 85
On-Premises Platforms, navigating 96-99
OpenTelemetry 289

P

Paging Hierarchy 299, 300
Performance Engineering 2-4
Performance Engineering, architecture 11-15
Performance Engineering, complexity 16
Performance Engineering, importance
bad customer, experience 18
box, testing 24, 25
code, reviews 25
IT Platform, complexities 21
IT Role, utilizing 22, 23
process, perspective 20
revenue, generating 20
usability, studies 19
user experience, tracking 18
user, optimizing 19

Performance Engineering, key points
budgets, optimizing 9, 10
data/page, weights 8, 9
internet service, providers 6, 7
network bandwidth 5
network speed 5, 6
revenue benefits, archiving 10, 11
Performance Tester 28
Performance Tester, impact
capacity planners 35
quality engineer 35
site reliability Engineer 35
software engineer 34
Performance Tester, observability 29
Performance Tester, tools 36-38
Performance Tester, types
Capacity 32
Performance Architect 34
Performance Load 31, 32
Platform Architect 33, 34
Product Architect/Business Analyst 33
Soak 30
Spike 30
Pervasive Design 263
Pervasive Design, capabilities
Cloud Database, configuring 266
Data Security, preventing 264
Data, syncing 265
HTML 5, advantages 263
Lightweight, optimizing 265
UI Tasks, optimizing 265
Product Development 73
Product Development, aspects
JMH, optimizing 76
Level Automation 77-80
PDD 75
Product Development, lifecycle
continuous, testing 74
Observability, monitoring 75
PDD, using 74
performance, environments 74
performance, latencies 74
requirement, competing 74
Project Delivery Resources 57

Q

Quality Gating 152-154
Quality Gating, aspects 152
Quality Gating, benefits
automate, auditing 154
flexibility, usability 154
Progressive Web Apps (PWA) 154
Quality Gating, disadvantages 153

R

RDBMS 236
RDBMS, tips
keys 237
performance, issues 236, 237
SQL, optimizing 237-240
Rendering Workflow 171-175
ResultSet 225
ResultSet, factors
Close/Nullify 225
database cursor 226
size, fetching 225

S

Scaling Cloud 131
Scaling Cloud, architecture
  Caching 140-142
  Database, optimizing 134
  EDA 132, 133
  Kubernetes 134
  Micro-Services 131
Scaling Cloud, methods 131
Service Level Agreement
    (SLA) 148
Service Level Indicators
    (SLI) 148
Service Level Objective
    (SLO) 149
Site Reliability Engineering
    (SRE) 68
SOA/Microservices,
    comparing 257, 258
SOA Service, predictions
  Cloud Services,
    managing 305
  delivery processes,
    improving 303
  device farms 302
  frontend performance,
    optimizing 302
  native compilation,
    preventing 304
  NoSQL 303
  Observability,
    analyzing 304
  Virtual IDE,
    configuring 305
  web frameworks,
    utilizing 304
Software Architecture 248
Software Architecture, types
  Client/Server 259
  Data Center Architecture
    (DCA) 259
  Docker 255, 256
  Enterpriser Service Bus (ESB) 256
  Event Driven Architecture
    (EDA) 258, 259
  Layered N-Tier 259-262
  Microservices 250-254
  Monolithic 249, 250
  Service Orchestration
    (SOA) 257
  Service Oriented 256
Software Development 40, 41
Software Development,
    innovation
  resources, staying 59, 60
  small initially, process 59
Software Development,
    methodologies
  IT Agile 52
  IT Iterative 51
  IT Waterfall 48
Software Development,
    outcomes
  Education 41
  individual roles,
    incorporating 43
  metrics, establishing 44
  ownership, encouraging 43
  performance goals,
    establishing 42
  performance, measuring 43
  performance, reviews 42, 43
Software Development,
    performance 101, 102
SRE, concepts

capacity, planning 70
incident, managing 71
infrastructure, managing 69
monitor/alerting 70
resiliency, availability 71
risk, assessments 71
Service Level, establishing 69
Stored Procedures 241, 242
Stored Procedures, benefits
efficient 241
reusable 241
traffic, reduction 241
String 220

T

Total Quality Management (TQM) 60
Transaction Script Data Pattern 226
Transaction Script Data Pattern, points
memory, consumption 226
performance 226
simplicity 226

V

VisualVM, tools
arrays 197
class instances 197
objects 197
strings 197

W

Webpack 9
Web Services, tuning 201-203
Workload Model 80
Workload Model, steps 81
WPO 159
WPO, benefits
brand, reputation 158
conversion rates, increasing 159
revenue, improving 158
search engine, ranking 159
security, resistance 159
user, experience 159
website speed, competitors 159
WPO, key points
Core Web Vitals 163, 164
Cross Browser/Device, testing 162, 163
Geo-Location, analyzing 162
IT Platform, optimizing 160, 161
TTL, optimizing 159, 160
Waterfall, preventing 161

Z

ZGC 192
ZGC, advantage 192
ZGC Performance, enhancing 193, 194